FAA

Fractal Academy of Analytics

SAS Training

Features and Understanding of PROC SQL

Agenda

- About SQL
- Data manipulation language
  - ✓ Select statement and its options
  - ✓ Editing data
- Data definition language
  - ✓ Create table
  - ✓ Create view
- Merging & appending databases
  - ✓ Joins
  - ✓ Append
- Transforming data base
  - ✓ Summarizing data
  - ✓ Case when statement
- Sub query

Structured Query Language (SQL)

- Structured Query Language
- Developed by IBM in the early 1970’s
- From the 70’s to the late 80’s there were different types of SQL, based on different databases.
- In 1986 the first unified SQL standard (SQL-86) was created.
- In 1987 database interface for SQL was added to the Version 6 Base SAS package
- A “language within a language”

Proc Sql - Terminology

SAS Data Step	Proc SQL
Dataset	Table
Variables	Column
Observation	Row
Append	Union
Merge	Join

What Can SQL do?

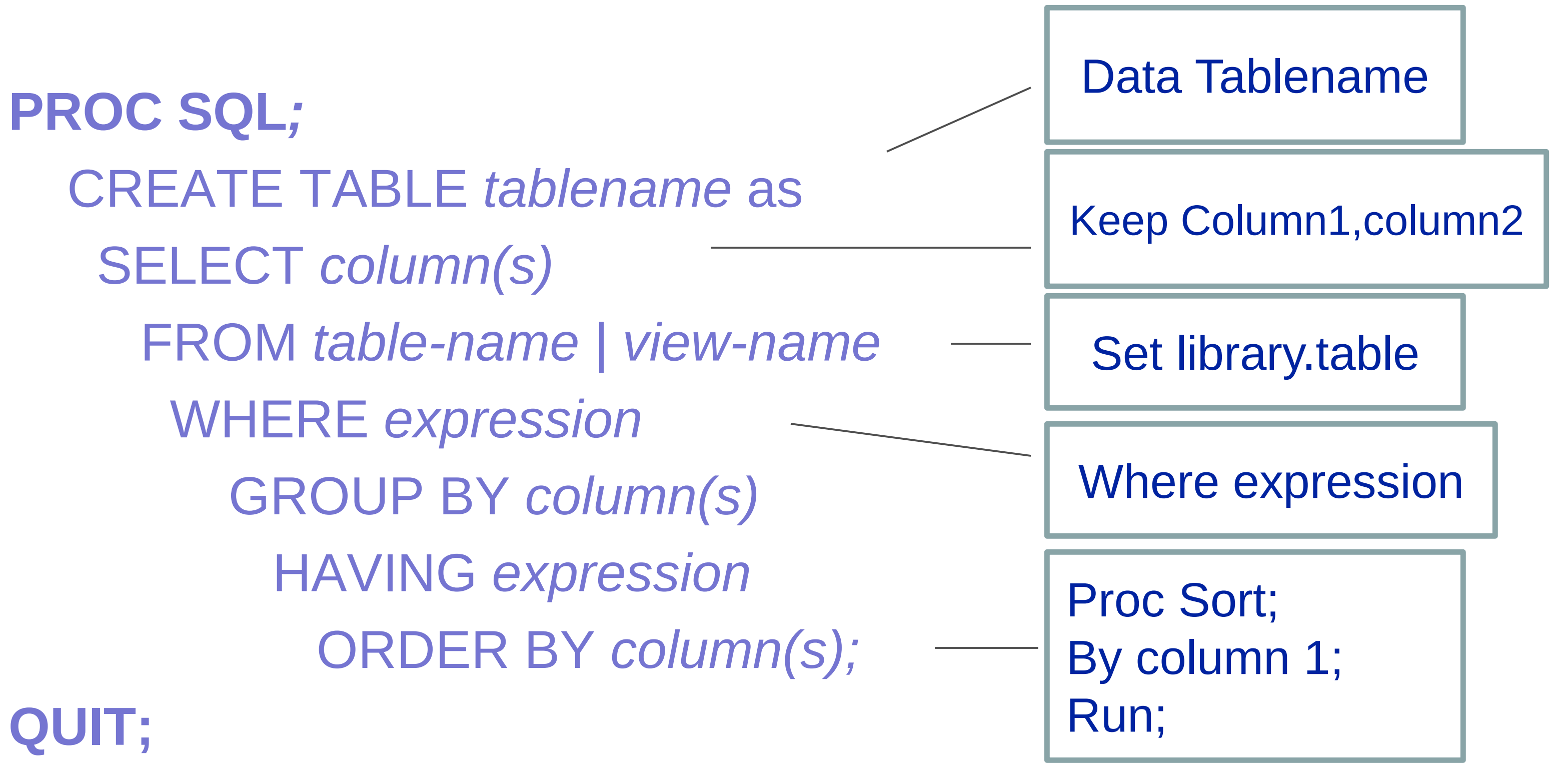
Feature	Statements used in Proc SQL
Summarize & Sort data	Where, Group by, Having, Order by and NC - SQL functions
Create tables & view	Create table, Create View
Update & Delete records	Update…Set…Where, Delete, Alter
Variable Transformations	Drop, Rename, NC - SQL functions,
Appending, Merging Datasets	Joins, Union
Subset data	Where, Case… when,
Insert observation values	Insert….Into, Insert…select

SQL DML/DDL

- The query and update commands form the Data manipulation language(DML) part of SQL:
  - **SELECT** - extracts data from a database
  - **UPDATE** - updates data in a database
  - **DELETE** - deletes data from a database
  - **INSERT INTO** - inserts new data into a database
  - **DISTINCT** - select distinct data from database

- The most important Data definition language(DDL) statements in SQL are:
  - **CREATE TABLE** - creates a new table
  - **ALTER TABLE** - modifies a table
  - **DROP TABLE** - deletes a table

Syntax



```
PROC SQL;
   CREATE TABLE tablename as
     SELECT column(s)
        FROM table-name | view-name
          WHERE expression
              GROUP BY column(s)
                 HAVING expression
                    ORDER BY column(s);
QUIT;
```

Data Tablename

Keep Column1,column2

Set library.table

Where expression

Proc Sort;
By column 1;
Run;

The SELECT Statement

- **SELECT Syntax**

```
SELECT column_name(s)
    FROM table_name; QUIT;

AND

SELECT * FROM table_name;
```

- **SELECT DISTINCT Syntax**

```
SELECT DISTINCT column_name(s)
    FROM table_name;
```

- The simplest SQL code, need 3 statements
- By default, it will print the resultant query, use NOPRINT option to suppress this feature.
- Begin with ***PROC SQL***, end with ***QUIT***; not ***RUN;***
- Need at least one ***SELECT… FROM*** statement
- *DISTINCT* is an option that removes duplicate rows

Functions of SQL

- PROC SQL supports all the functions available to the SAS DATA step that can be used in a **Proc Sql** select statement
- Because of how SQL handles a dataset, these functions work over the entire dataset
- **Common Functions:**

✓ COUNT
✓ DISTINCT
✓ MAX
✓ MIN
✓ SUM
✓ AVG
✓ VAR
✓ STD
✓ STDERR
✓ NMISS
✓ RANGE
✓ SUBSTR
✓ LENGTH
✓ UPPER
✓ LOWER
✓ CONCAT
✓ ROUND
✓ MOD

- PROC SQL does not support LAG, DIF, and SOUND functions.

The WHERE Clause

- The WHERE clause is used to extract only those records that fulfill a specified criterion.
- WHERE Syntax

```
SELECT column_name(s)
  FROM table_name
  WHERE column_name operator value;
```

- SQL uses single quotes around text values (most database systems will also accept double quotes)
- Numeric values should not be enclosed in quotes.

Operators Allowed in the WHERE Clause

Operator	Description
Eq	Equal
Ne	Not equal
Gt	Greater than
Lt	Less than
Ge	Greater than or equal
Le	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns

- **Note:** Not equal to <> operator may be written as !=

- ## The AND & OR Operators
  - The AND operator displays a record if both the first condition and the second condition is true.
  - The OR operator displays a record if either the first condition or the second condition is true.

Some examples

- Using **AND** in where clause

```
SELECT * FROM Persons
   WHERE FirstName='Tove'
   AND LastName='Svendson‘;
```

- Using **OR** in where clause

```
SELECT * FROM Persons
   WHERE FirstName='Tove'
   OR FirstName='Ola‘;
```

- Combining **AND & OR** in where clause

```
SELECT * FROM Persons WHERE
   LastName='Svendson'
   AND (FirstName='Tove' OR FirstName='Ola');
```

Like Operator

- The **LIKE** operator is used to search for a specified pattern in a column.

```
SELECT column_name(s)
    FROM table_name
    WHERE column_name LIKE pattern;
```

- **SQL Wildcards**

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for exactly one character

- Using the **% Wildcard**

```
SELECT * FROM Persons
    WHERE City LIKE 'sa%‘;
```

- Using the _ **Wildcard**

```
SELECT * FROM Persons
    WHERE FirstName LIKE '_la‘;
```

More Operators

- The **IN** Operator
  - The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT column_name(s)
    FROM table_name
    WHERE column_name IN (value1,value2,...);
```

- The **BETWEEN** Operator
  - The BETWEEN operator selects a range of data between two values.

```
SELECT column_name(s)
    FROM table_name
    WHERE column_name
    BETWEEN value1 AND value2;
```

- **Note:** The BETWEEN operator is treated differently in different databases!

The ORDER BY Keyword

- The **ORDER BY** keyword is used to sort the result-set by a specified column
- The **ORDER BY** keyword sort the records in ascending order by default
- If you want to sort the records in a descending order, you can use the **DESC** keyword.

```
SELECT column_name(s)
    FROM table_name
    ORDER BY column_name(s) ASC|DESC;
```

```
SELECT * FROM Persons
    ORDER BY LastName;
```

```
SELECT first_name, last_name FROM Persons
    ORDER BY 1;
```

The GROUP BY Keyword

- The **GROUP BY** clause can be used to summarize or aggregate data.

- Summary functions (also referred to as aggregate functions) are used on the SELECT statement for each of the analysis variables:

```
PROC SQL;
SELECT STATE, SUM(Column name) AS Column alias name
FROM table
GROUP BY STATE;
QUIT;
```

- Other summary functions available are the AVG/MEAN, COUNT/FREQ/N, MAX, MIN, NMISS, STD, SUM, and VAR.
- This capability Is similar to PROC SUMMARY with a CLASS statement.

The HAVING Clause

- The **HAVING** clause works with the GROUP BY clause to restrict the groups in a query’s results based on a given condition.
- PROC SQL applies the **HAVING** condition after grouping the data and applying aggregate functions.

```
PROC SQL;

SELECT STATE, SUM(Column name) AS Column alias name

FROM Table name

GROUP BY STATE

HAVING STATE IN (‘ABC’, ‘XYZ’);

QUIT;
```

The UPDATE Statement

- The **UPDATE** statement is used to update existing records in a table.

```
UPDATE table_name
    SET column1=value, column2=value2,...
    WHERE some_column = some_value;
```

- **SQL UPDATE Warning**
  - The WHERE clause specifies which record or records that should be updated.
  - **If you omit the WHERE clause, all records will be updated!**

Editing Data

– Update Observations

/*Updating Observation*/

```
PROC SQL NOPRINT;
UPDATE table
  SET Column name= value
  WHERE Column name=Value ;
QUIT;
```

- UPDATE … SET… WHERE
- Find the observation and set new value
- If more than one observations satisfies the condition, all are updated with the new data in SET statement

Editing Data

– Renaming Rows & No. of Observations

/*Renaming rows*/

```
PROC SQL;
Create table table name
(rename =(Column name current=
Column name future)) as
     FROM Table
  WHERE Column name LE value;
QUIT;
```

/*Observation selection*/

```
PROC SQL;
 CREATE TABLE table name
      AS
 SELECT *
  FROM table (obs=value);
QUIT;
```

- Renaming columns can be done in CREATE Statement
- Selection of Number of Observation is done in the FROM Statement

The DELETE Statement

- The **DELETE** statement is used to delete rows in a table.

```
DELETE FROM table_name
    WHERE some_column=some_value
```

- **Delete All Rows**
  - It is possible to delete all rows in a table without deleting the table.
  - This means that the table structure, attributes, and indexes will be intact:

DELETE FROM table_name;

OR

*DELETE * FROM table_name;*

Editing Data

– Deleting rows and Dropping columns

```
/*Deleting rows*/

PROC SQL;
DELETE
  FROM table
  WHERE column name LE  Value;
QUIT;
```

```
/*Dropping variables*/

PROC SQL;
 CREATE TABLE table name
      (DROP=column name) AS
 SELECT *
  FROM table;
QUIT;
```

- Deleting columns can be done in SELECT or in DROP on created table

Creating New Data

- Create Table

```
PROC SQL;
 CREATE TABLE TABLE NAME as
 SELECT column names
 FROM table
 WHERE Colum name CONTAINS value;
QUIT;
```

- CREATE TABLE … AS can always be in front of SELECT … FROM statement to build a sas file.
- In SELECT, the results of a query are converted to an output object (printing). Query results can also be stored as data. The CREATE TABLE statement creates a table with the results of a query.

Creating New Data

- Create View

```
PROC SQL;
 CREATE VIEW G_MOVIES as
                SELECT Title, Author, ISBN, Price
                FROM Library. Books
     WHERE Price = 235
     ORDER BY Price;
 SELECT  *  FROM G_MOVIES;
QUIT;
```

- First step-creating a view, no output is produced; then display the desired output results
- Use ; to separate two block of code inside of proc sql
- When a table is created, the query is executed and the resulting data is stored in a file. When a view is created, the query itself is stored in the file. The data is not accessed at all in the process of creating a view.

The INSERT INTO Statement

- The **INSERT INTO** statement is used to insert a new row in a table.
- It is possible to write the **INSERT INTO** statement in two forms.
  - The first form doesn't specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
   VALUES (value1, value2, value3,...);
```

  - The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...)
   VALUES (value1, value2, value3,...);
```

Editing Data

– Insert Observations

```
PROC SQL NOPRINT;
INSERT INTO MFE.CUSTOMERS(a,b)
  VALUES(1 'Peng‘, 2 ’rid’,3 ‘sam’);
INSERT INTO MFE.CUSTOMERS
  SET Cust_no=2,Name='Sasha';
QUIT;
```

- There are two ways of inserting observations into a table. Data type should be the same
- VALUES( ) new values are separated by space
- SET column name = newly assigned values, delimited by commas

Transforming Data

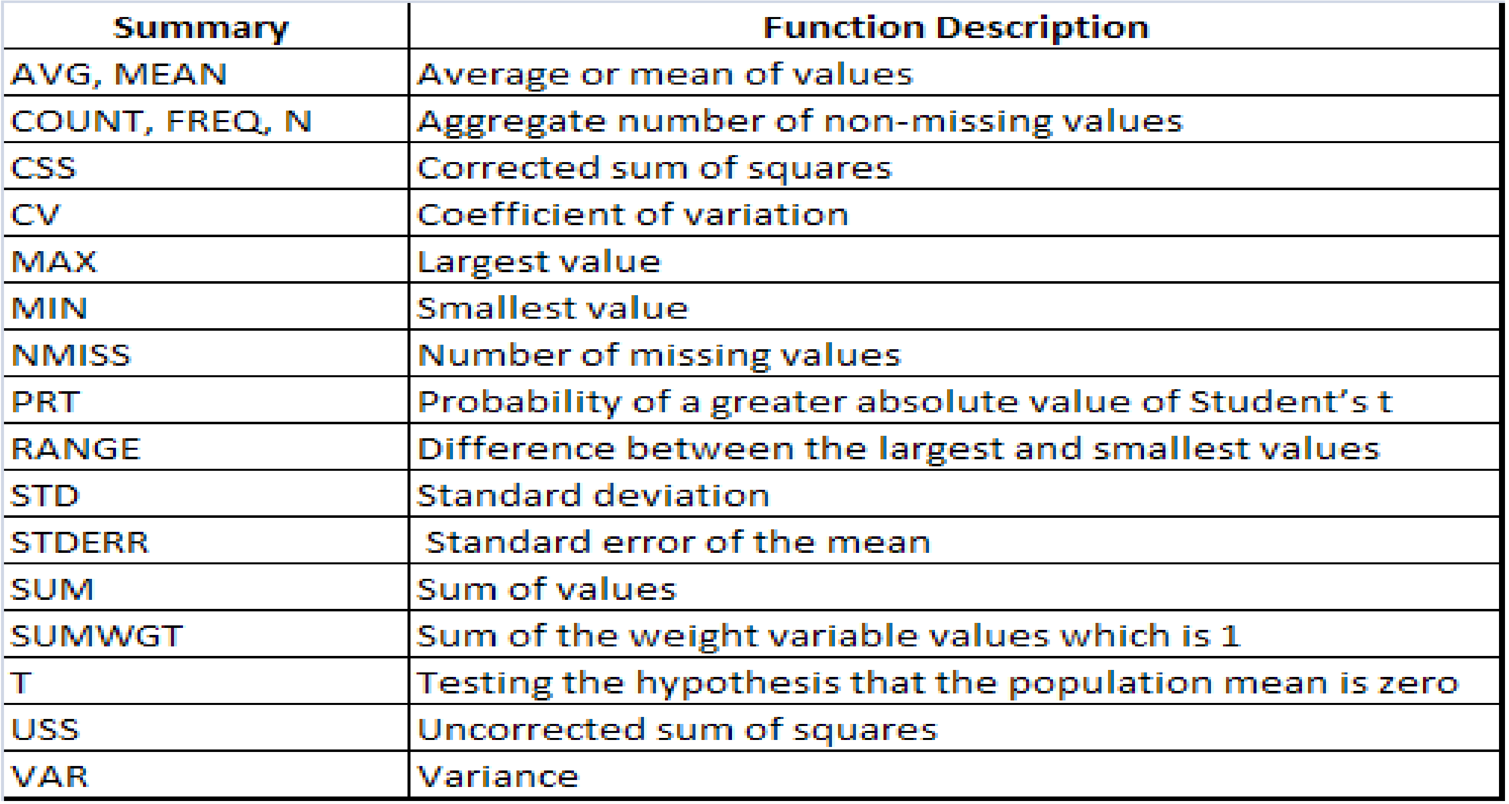
- Summarizing Data using SQL functions

```
PROC SQL;
  SELECT *,
                COUNT(Title)  AS notitle,
                MAX(Price)      AS Expensive,
                MIN(Price)       AS Cheapest,
                SUM(Price)   AS Total_Cost,
                NMISS(Author) AS nomissing
  FROM Library.Books
  GROUP BY Author;
QUIT;
```

- Simple summarization functions available
- All function can be operated in Groups
- Re-merging summary statistics with Original data

Summarizing Data

- It provides a number of useful summary (or aggregate) functions to help perform calculations, descriptive statistics, and other aggregating operations in a SELECT statement or HAVING clause.



Summary	Function Description
AVG, MEAN	Average or mean of values
COUNT, FREQ, N	Aggregate number of non-missing values
CSS	Corrected sum of squares
CV	Coefficient of variation
MAX	Largest value
MIN	Smallest value
NMISS	Number of missing values
PRT	Probability of a greater absolute value of Student's t
RANGE	Difference between the largest and smallest values
STD	Standard deviation
STDERR	Standard error of the mean
SUM	Sum of values
SUMWGT	Sum of the weight variable values which is 1
T	Testing the hypothesis that the population mean is zero
USS	Uncorrected sum of squares
VAR	Variance

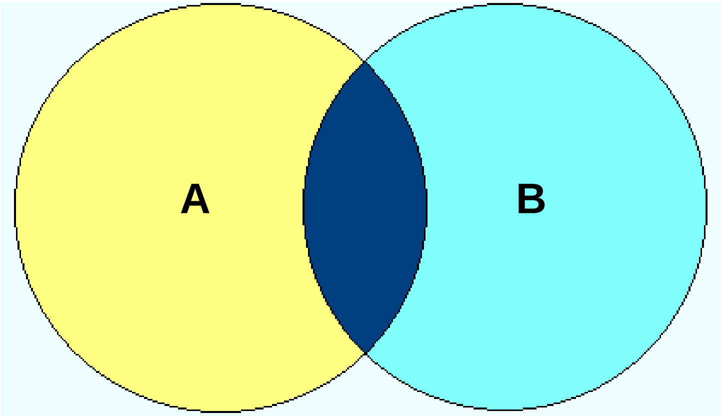
Different SQL Joins

- **Inner JOIN**:
  - Return rows when there is at least one match in both tables
- **LEFT JOIN**:
  - Return all rows from the left table, even if there are no matches in the right table
- **RIGHT JOIN**:
  - Return all rows from the right table, even if there are no matches in the left table
- **FULL JOIN**:
  - Return rows when there is a match in one of the tables

SQL Joins
– Inner Join

- The **INNER JOIN** keyword return rows when there is at least one match in both tables.

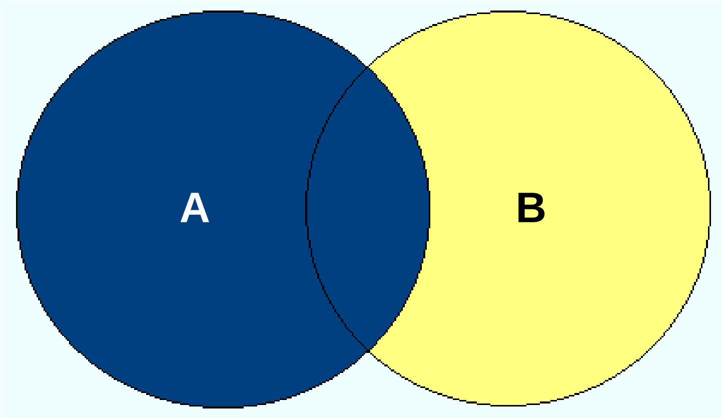
```
SELECT column_name(s)
    FROM table_name1
    INNER JOIN table_name2
    ON table_name1.column_name=table_name2.column_name;
```



SQL Joins
– Left Join

- The **LEFT JOIN** keyword returns all rows from the left table (table_name1), even if there are no matches in the right table (table_name2).

```
SELECT column_name(s)
   FROM table_name1
   LEFT JOIN table_name2
   ON table_name1.column_name=table_name2.column_name;
```



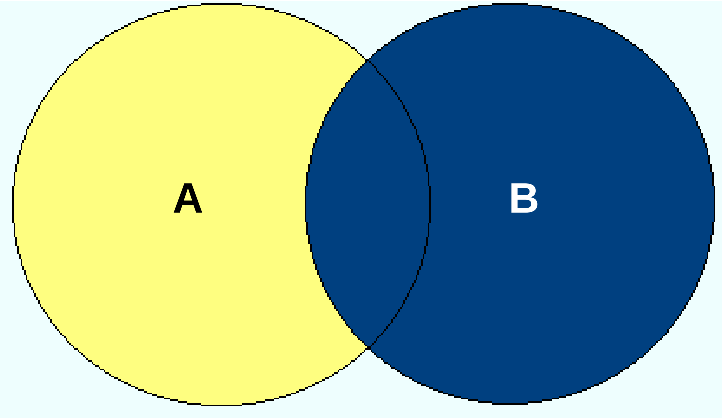
- **PS:** In some databases LEFT JOIN is called LEFT OUTER JOIN

SQL Joins

– Right Join

- The **RIGHT JOIN** keyword returns all the rows from the right table (table_name2), even if there are no matches in the left table (table_name1).

```
SELECT column_name(s)
   FROM table_name1
   RIGHT JOIN table_name2
   ON table_name1.column_name=table_name2.column_name;
```



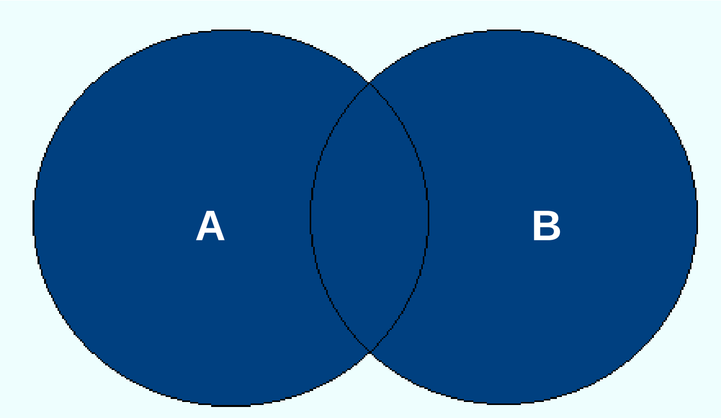
- **PS:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

SQL Joins

– Full Join

- The **FULL JOIN** keyword return rows when there is a match in one of the tables.

```
SELECT column_name(s)
   FROM table_name1
   FULL JOIN table_name2
   ON table_name1.column_name=table_name2.column_name;
```



SQL Joins

– Self Join & Cross join

Self join: A table joined with itself to produce more information

SYNTAX:

```
 SELECT column_name(s)
FROM table_name1 as a,  table_name1 as b
ON a.column_name=b.column_name;
```

Cross join: Cartesian product of two or more columns in one or more databases

SYNTAX:

```
 SELECT column_name(s)
FROM table_name1,  table_name2
ON tablename1.column_name=tablename2.column_name;
```

Union Operator

- The **UNION** operator is used to combine the result-set of two or more SELECT statements.
- **Remember :**
  - Each SELECT statement within the UNION must have the same number of columns
  - The columns must also have similar data types
  - The columns in each SELECT statement must be in the same order
- **Syntax**:

```
SELECT column_name(s) FROM table_name1
    UNION
    SELECT column_name(s) FROM table_name2;
```

- **PS:** The column names in the result-set of a UNION are always equal to the column names in the first SELECT statement in the UNION.

Union All

- The **UNION** operator selects **only distinct values** by default. To allow duplicate values, use UNION ALL.

- **Syntax**

```
SELECT column_name(s) FROM table_name1
    UNION ALL
    SELECT column_name(s) FROM table_name2;
```

Union Example

E_ID	E_Name
01	Hansen, Ola
02	Svendson, Tove
03	Svendson, Stephen
04	Pettersen, Kari

E_ID	E_Name
01	Turner, Sally
02	Kent, Clark
03	Svendson, Stephen
04	Scott, Stephen

```
SELECT E_Name FROM Employees_Norway
    UNION
    SELECT E_Name FROM Employees_USA;
```

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Scott, Stephen

Union All Example

E_ID	E_Name
01	Hansen, Ola
02	Svendson, Tove
03	Svendson, Stephen
04	Pettersen, Kari

E_ID	E_Name
01	Turner, Sally
02	Kent, Clark
03	Svendson, Stephen
04	Scott, Stephen

```
SELECT E_Name FROM Employees_Norway
    UNION ALL
    SELECT E_Name FROM Employees_USA;
```

E_Name
Hansen, Ola
Svendson, Tove
Svendson, Stephen
Pettersen, Kari
Turner, Sally
Kent, Clark
Svendson, Stephen
Scott, Stephen

Except option

```
PROC SQL;
   SELECT * FROM Library.Books
     EXCEPT
       SELECT * FROM Library.Order_Date;
QUIT;
```

- The EXCEPT operator produces (from the first table-expression) an output table that has unique rows that are not in the second table-expression

Identifying Duplicates

```
 PROC SQL;
    CREATE TABLE Author as
      SELECT Author,
                Sum(Price) as Price
          FROM Library.Books
            GROUP BY Author
             HAVING COUNT(*) GE 2
             ORDER BY Price ;
QUIT;
```

Case Logic
- Reassigning/ Re-categorize

- The **CASE** expression selects values if certain conditions are met. A CASE expression returns a single value that is conditionally evaluated for each row of a table (or view).

```
PROC SQL;
SELECT Title, Author,
         CASE
               WHEN Author = ‘Cody'  THEN ‘General'   ELSE  'Other'
         END AS level
 FROM Library. Books;
QUIT;
```

- The order of each statement is important
- CASE …END AS should in between SELECT and FROM
- Note there is , after the variables you want to select
- Use WHEN … THEN ELSE… to redefine variables
- Rename variable from “Author” to “level”

Case Logic

- Sum/ Count

```
PROC SQL;
SELECT
        SUM (CASE WHEN Author= 'G' THEN 1 END) as General,
        SUM( CASE WHEN Author=‘G’ THEN Price END) as Price,
         Count (*) as Books
    FROM Library. Books
Group by Author;
QUIT;
```

The Count () function returns the number of rows as defined by the Group Statement

Correlated Sub-Query

```
PROC SQL;
  CREATE TABLE Corr_Query AS
   SELECT DISTINCT Author, Price
    FROM Library.Books as B
     WHERE ‘1590473337’ IN
      (SELECT Order_Date
       FROM Library.Order_Date as O
         WHERE B.ISBN=O.ISBN)
           ORDER BY Price;
QUIT;
```

Non-Correlated Sub-Query

```
PROC SQL;
CREATE TABLE NON_CORR as
     SELECT Author,
             Avg(Price) as Avg_Price
                 FORMAT = dollar11.2
      FROM Library.Books
      GROUP BY Author
      HAVING Avg(Price) >
              (SELECT Avg(Price)
               FROM Library.Books);
   QUIT;
```

Using calculated field in SQL

```
PROC SQL;
CREATE TABLE new as
SELECT Author ,Price  ,(Price*10) as Value
FROM Library.Books
WHERE calculated Value > 5000
GROUP by Author
HAVING Mean(Value) gt 54;
QUIT;
```

Validation of Syntax

```
PROC SQL;
    VALIDATE
   SELECT *
     FROM Library.Books
       WHERE Price= 780;
QUIT;
```

- Help in Troubleshooting and debugging the SQL Queries.
- It is Specified in Conjunction with a SELECT Statement.
- The appropriate message is displayed on the SAS log to indicate whether coding problems exist.

Validation of Syntax

```
PROC SQL noexec;
    SELECT *
       FROM Library.Book
          WHERE Price= 780;
QUIT;
```

- Help in Troubleshooting and debugging the SQL Queries.
- It is specified in the Procedure Step
- The appropriate message is displayed on the SAS log to indicate whether coding problems exist.

Display the Contents of Dataset

```
PROC SQL;
  Describe table Library.Books;
QUIT;
```

- Help in Displaying the Variables Name with format and Informat of the Variables
- Just like the Proc Contents
- Display the result only in Log

Multiple Dataset in Single Proc Sql

```
PROC SQL;
 Create table Obs as
  SELECT *
   FROM Library.Books (obs=4);

   Create table Author as
   Select Author
   from Library.Books
   where Author='Cody';

QUIT;
```

Errors

- Syntax error, expecting one of the following: !, !!, &, (, *, **, +, ',', -, /, <, <=, <>, =, >, >=, ?, AND, BETWEEN, CONTAINS, EQ, FROM, GE, GT, LE, LIKE, LT, NE, OR, ^=, |, ||, ~=.
- ERROR: Libname LIBRARY is not assigned.
- ERROR: File WORK.BOOKS.DATA does not exist.
- WARNING: This SAS global statement is not supported in PROC SQL. It has been ignored.
- ERROR : Syntax error, statement will be ignored.
- WARNING: Data too long for column "COMMENT"; truncated to 124 characters to fit.
- WARNING: Variable N_Time already exists on file WORK.LD50_PARMS.

Thank You!!!