A Crash Course in SQL

Table of Contents

1. Introduction

In this note, we will briefly introduce the concepts of relational databases. The focus will be the syntax of SQL and how to run SQL in SAS.

There are diverse ways to run SQL. For example, you can use different SQL related packages and libraries in either R or Python to run SQL. There are several reasons for choosing SAS PROC SQL to query relational data tables: (1) the base SAS programming class is a required course in the program; (2) SAS PROC SQL can authentic SQL when it is connected to a relational database. (3) SAS 9.4 and newer version implemented a new procedure FEDSQL that allows SAS to connect multiple relational databases at same time.

In this course, we primarily focus writing SQL through SAS PROC SQL to extract information from relational data table. Whenever your data task can be performed with SQL, please avoid regular SAS data steps.

2. Concepts of Relational Databases and SQL

This section illustrates the basics of data, database, database management system in a non-technical approach.

2.1. Data and Database

Data is recorded information that has been translated into a form that is efficient for movement or processing. For example, a data about a student may include information like name, unique id, age, address, education, etc.

A database is an organized collection of data, which is generally stored and accessed electronically from a computer system. In simple words, we can say a database in a place where the data is stored. The best analogy is the library. The library contains a massive collection of books of different genres, here library is database and books are the data.

The database can be classified broadly into the following groups:

- Centralized database
- Distributed database
- NoSQL database
- Operational database
- Relational database
- Cloud database
- Object-oriented database
- Graph database

We will be focusing more on the relational database which uses SQL for querying its tables.

A **table** in a relational database is nothing but a collection of data in a tabular way. It consists of *columns* and *rows*. The table contains data elements also known as values

using a model of vertical columns and horizontal rows. The point of intersection of a row and a column is called *a CELL*. A table can have any number of rows but should have a specified number of columns.

All tables in a relational database contain records and all records are identified by a field containing a unique value (also called *primary key*). Every table shares at least one field with another table in one-to-one, one-to-many or many-to-many relationships. This allows the user to access the data in many ways.

2.2. Database Management Systems (DBMS)

A Database Management System (DBMS) is software designed to store, retrieve, define, and manage data in a database. There are several different database management systems for working with relational data such as MySQL, PostgreSQL, MS Access, Filemaker Pro. The only things that will differ are the details of exactly how to import and export data and the details of data types.

2.3. What we will NOT cover in this course

We have introduced the basics of database and database management system. But we will not use any DBMS in this course. Instead we use several relational tables and SAS PROC SQL to practice SQL.

The following topics will **not** be covered in this class.

- Database design
- NoSQL databases
- How to connect different software system to different DBMSs.
- Updating databases

2.4. SQL Data Types

Before we move to SQL coding, we introduce the basic data types.

Data type	Access	SQLServer	Oracle	MySQL	PostgreSQL
boolean	Yes/No	Bit	Byte	N/A	Boolean
integer	Number (integer)	Int	Number	Int / Integer	Int / Integer
float	Number (single)	Float / Real	Number	Float	Numeric
currency	Currency	Money	N/A	N/A	Money
string (fixed)	N/A	Char	Char	Char	Char
string (variable)	Text (<256) / Memo (65k+)	Varchar	Varchar / Varchar2	Varchar	Varchar
binary object OLE Object Memo Binary	Varbinary (<8K)	Image (<2GB) Long	Raw Blob	Text Binary	Varbinary

3. Basic queries

First, we use the following three data sets that are related to each other with some key. These data files are available on the course web page. Let's start by using the **surveys** table. Here we have data on every individual that was captured at the site, including when they were captured, what plot they were captured on, their species ID, sex and weight in grams.

- surveys.csv

A	B	C	D	E	F	G	H	I
record_id	month	day	year	plot_id	species_id	sex	hindfoot_leng	weight
85	8	20	1977	23	DM	F	35	41
86	8	20	1977	18	DM	F	33	40
87	8	20	1977	5	PF	F	11	9
88	8	20	1977	18	DM	F	35	45
89	8	20	1977	12	PP	F	20	15
90	8	20	1977	18	DM	M	35	29
91	8	20	1977	11	DS	F	50	
92	8	20	1977	6	DM	M	35	39
93	8	20	1977	18	DM			42
94	8	20	1977	18	DM	F	36	43

- species.csv

species_id	genus	species	taxa
AB	Amphispiza	bilineata	Bird
AH	Ammospermophilus	harrisi	Rodent
AS	Ammodramus	savannarum	Bird
BA	Baiomys	taylori	Rodent
CB	Campylorhynchus	brunneicapillus	Bird
CM	Calamospiza	melanocorys	Bird
CQ	Callipepla	squamata	Bird
CS	Crotalus	scutalatus	Reptile
CT	Cnemidophorus	tigris	Reptile

- plots.csv

plot_id	plot_type
1	Spectab exclosure
2	Control
3	Long-term Krat Exclosure
4	Control
5	Rodent Exclosure
6	Short-term Krat Exclosure
7	Rodent Exclosure
8	Control
9	Spectab exclosure

Pay attention to the variables **species_id** and **plot_id** in the survey table. These keys are used to make the connection to the other two tables.

3.1. Types of Questions May Be Asked From the Data Tables

What information is contained in each file? Specifically, people may ask questions like:

- How has the hindfoot length and weight of Dipodomys species changed over time?
- What is the average weight of each species, per year?
- What information can I learn about Dipodomys species in the 2000s, over time?

To answer the questions described above, we'll need to do the following basic data operations:

- select subsets of the data (rows and columns)
- group subsets of data
- do math and other calculations
- combine data across spreadsheets

Putting our data into a relational database and using SQL will help us achieve these goals.

We first load the three table to SAS (i.e., create three SAS data sets, you can consider the three SAS data sets as three data tables in a relational database).

```
LIBNAME sql "C:\STA551\w02\SQLCrashCourse";
/** loading three data files **/
PROC IMPORT OUT= SQL.survey
        DATAFILE= "C:\STA551\w02\surveys.csv"
        DBMS=CSV REPLACE;
        GETNAMES=YES;
        GUESSINGROWS = MAX;
        DATAROW=2;
RUN;

PROC IMPORT OUT= SQL.species
        DATAFILE= "C:\STA551\w02\species.csv"
        DBMS=CSV REPLACE;
        GETNAMES=YES;
        GUESSINGROWS = MAX;
        DATAROW=2;
RUN;


PROC IMPORT OUT= SQL.plots
        DATAFILE= "C:\STA551\w02\plots.csv"
        DBMS=CSV REPLACE;
       GETNAMES=YES;
       GUESSINGROWS = MAX;
       DATAROW=2;
RUN;
```

3.2. Writing Your First Query

Let's start by using the **surveys** table. Here we have data on every individual that was captured at the site, including when they were captured, what plot they were captured on, their species ID, sex and weight in grams.

Let's write an SQL query that selects only the year column from the surveys table in the following.

```
*Table view;
PROC SQL;
SELECT year
FROM sql.survey;
QUIT;
```

We have capitalized the words SELECT and FROM because they are SQL keywords. SQL is case insensitive, but it helps for readability, and is good style.

If we want more information, we can just add a new column to the list of fields, right after `SELECT`:

```
* create a new table;
PROC SQL;
CREATE TABLE sql.YMD AS
SELECT year, month, day
```

```
FROM sql.survey;
QUIT;
```

Or we can select all of the columns in a table using the wildcard *

```
* create a new table: select all variables;
PROC SQL;
CREATE TABLE sql.survey_all AS
SELECT *
FROM sql.survey;
QUIT;
```

3.3. Unique values

If we want only the unique values so that we can quickly see what species have been sampled we use `DISTINCT`

```
* one variable;
PROC SQL;
SELECT DISTINCT year
FROM sql.survey;
QUIT;
```

If we select more than one column, then the distinct pairs of values are returned

```
* two variable;
PROC SQL;
SELECT DISTINCT year, species_id
FROM sql.survey;
QUIT;
```

3.4. Calculated values

We can also do calculations with the values in a query. For example, if we wanted to look at the mass of each individual on different dates, but we needed it in kg instead of g we would use

```
* Create a view;
PROC SQL;
SELECT year,
        month,
        day,
       weight/1000.0
FROM sql.survey;
QUIT;

* Create a table and define a new variable
  based on the calculated values;
PROC SQL;
CREATE TABLE sql.Add_new_var AS
SELECT year,
    month,
    day,
```

```
    weight/1000.0 AS wt_kilo
FROM sql.survey;
QUIT;
```

When we run the query, the expression `weight / 1000.0` is evaluated for each row and appended to that row, in a new column. Expressions can use any fields, any arithmetic operators (`+`, `-`, `*`, and `/`) and a variety of built-in functions. For example, we could round the values to make them easier to read.

```
* Using function ROUND();
* CAUTION: ROUND() in PROC SQL is an SAS function!
* To keep 2 decimal places, SAS uses ROUND(number, 0.01)
                    SQL uses ROUND(number, 2);
PROC SQL;
SELECT plot_id,
        species_id,
        sex,
        weight,
        ROUND(weight / 1000.0, 0.01) /* ROUND() is a SAS function! */
FROM sql.survey;
QUIT;
```

3.5. Filtering

Databases can also filter data – selecting only the data meeting certain criteria. For example, let's say we only want data for the species *Dipodomys merriami*, which has a species code of DM. We need to add a `WHERE` clause to our query:

```
* subsetting by filtering - WHERE statement;
* Single condition;
PROC SQL;
SELECT *
FROM sql.survey
WHERE species_id='DM';
QUIT;
```

We can use more sophisticated conditions by combining tests with `AND` and `OR`. For example, suppose we want the data on *Dipodomys merriami* starting in the year 2000:

```
* Multiple conditions: AND;
PROC SQL;
SELECT *
FROM sql.survey
WHERE (year >= 2000) AND (species_id = 'DM');
QUIT;
```

Note that the parentheses are not needed, but again, they help with readability. They also ensure that the computer combines `AND` and `OR` in the way that we intend.

If we wanted to get data for any of the *Dipodomys* species, which have species codes `DM`, `DO`, and `DS`, we could combine the tests using OR:

```
* Multiple conditions: OR;
PROC SQL;
SELECT *
FROM sql.survey
WHERE (species_id = 'DM') OR (species_id = 'DO') OR (species_id = 'DS');
QUIT;
```

3.6. Special Keywords in WHERE Statement

Now, lets combine the above queries to get data for the 3 *Dipodomys* species from the year 2000 on. This time, let's use `IN` as one way to make the query easier to understand. It is equivalent to saying `WHERE (species_id = 'DM') OR (species_id = 'DO') OR (species_id = 'DS')`, but reads more neatly:

```
* use of keyword IN;
PROC SQL;
SELECT *
FROM sql.survey
WHERE (year >= 2000) AND (species_id IN ('DM', 'DO', 'DS'));
QUIT;
```

We started with something simple, then added more clauses one by one, testing their effects as we went along. For complex queries, this is a good strategy, to make sure you are getting what you want. Sometimes it might help to take a subset of the data that you can easily see in a temporary database to practice your queries on before working on a larger or more complicated database.

CAUTION: When the queries become more complex, it can be useful to add comments. Unlike in SAS, comments are started by "`/* long comments */`" and "`* short comment ;`" , In SQL, comments are started by `--`, and end at the end of the line.

3.7. Sorting

We can also sort the results of our queries by using `ORDER BY`. For simplicity, let's go back to the **species** table and alphabetize it by taxa.

First, let's look at what's in the **species** table. It's a table of the species_id and the full genus, species and taxa information for each species_id. Now let's order it by taxa.

```
* Ascending ordering - default;
PROC SQL;
SELECT *
FROM sql.species
ORDER BY taxa ASC;
QUIT;
```

The keyword `ASC` tells us to order it in Ascending order. We could alternately use `DESC` to get descending order.

```
* descending ordering;
```

```
PROC SQL;
SELECT *
FROM sql.species
ORDER BY taxa DESC;
QUIT;
```

We can also sort on several fields at once. To truly be alphabetical, we might want to order by genus then species.

```
* sorting multiple variables- nest sorting;
PROC SQL;
SELECT *
FROM sql.species
ORDER BY genus ASC, species ASC;
QUIT;
```

3.8. Order of Execution

Another note for ordering. **We don't actually have to display a column to sort by it**. For example, let's say we want to order the birds by their species ID, but we only want to see genus and species.

```
*Clauses are written in a fixed order: SELECT, FROM, WHERE, then ORDER BY. ;
PROC SQL;
SELECT genus, species
FROM sql.species
WHERE taxa = 'Bird'
ORDER BY species_id ASC;
QUIT;
```

We can do this because **sorting occurs earlier in the computational pipeline** than field selection.

The computer is basically doing this:

1. Filtering rows according to `WHERE`
2. Sorting results according to `ORDER BY`
3. Displaying requested columns or expressions.

Clauses are written in a fixed order: `SELECT`, `FROM`, `WHERE`, then `ORDER BY`. It is possible to write a query as a single line, but for readability, we recommend putting each clause on its own line.

4. Aggregation

4.1. COUNT and GROUP BY

Aggregation allows us to combine results by grouping records based on value and calculating combined values in groups.

Aggregation allows us to **combine results by grouping records** based on value and **calculating combined values in groups**.

Let's go to the surveys table and find out **how many individuals** there are. Using the wildcard simply counts the number of records (rows)

```
*no group - Using the wildcard simply counts the number of records (rows);
PROC SQL;
SELECT COUNT(*)
FROM sql.survey;
QUIT;
```

We can also find out how much all of those individuals weigh.

```
* calculate the sum of a numerical variable;
PROC SQL;
SELECT COUNT(*) ,
        SUM(weight)
FROM sql.survey;
QUIT;
```

There are many other aggregate functions included in SQL including `MAX`, `MIN`, and `AVG`.

Now, let's see how many individuals were counted in each species. We do this using a `GROUP BY` clause

```
*summary statistics within subgroups - GROUP BY clause;
* This is equalent to a univeriable frequency table;
PROC SQL;
SELECT species_id,
        COUNT(*)
FROM sql.survey
GROUP BY species_id;
QUIT;
```

`GROUP BY` tells SQL what field or fields we want to use to aggregate the data. If we want to group by multiple fields, we give `GROUP BY` a comma separated list.

4.2. The HAVING keyword

In the previous section, we have seen the keywords `WHERE`, allowing to filter the results according to some criteria. SQL offers a mechanism to filter the results based on aggregate functions, through the `HAVING` keyword.

For example, we can adapt the last request we wrote to only return information about species with a count higher than 10:

```
* conditioning on species size;
PROC SQL;
SELECT species_id,
    COUNT(species_id)
FROM sql.survey
GROUP BY species_id
HAVING COUNT(species_id) > 10;
QUIT;
```

```
* conditioning on species size - a better code;
PROC SQL;
SELECT species_id,
    COUNT(species_id) AS species_size
FROM sql.survey
GROUP BY species_id
HAVING species_size > 10;
QUIT;
```

The `HAVING` keyword works exactly like the `WHERE` keyword, but uses aggregate functions instead of database fields.

Note that in both queries, `HAVING` comes *after* `GROUP BY`. One way to think about this is: the data are retrieved (`SELECT`), can be filtered (`WHERE`), then joined in groups (`GROUP BY`); finally, we only select some of these groups (`HAVING`).

4.3. Ordering aggregated results.

We can order the results of our aggregation by a specific column, including the aggregated column. Let's count the number of individuals of each species captured, ordered by the count

```
* sorting aggregated variables -- This DOES NOT work!
* Summary functions are restricted to the SELECT and HAVING clauses only;
PROC SQL;
SELECT species_id,
        COUNT(*)
FROM sql.survey
GROUP BY species_id
ORDER BY COUNT(species_id);
QUIT;
```

```
* sorting aggregated variables;
* use the new name in the ORDER BY clause;
PROC SQL;
SELECT species_id AS subtotal,
        COUNT(*)
FROM sql.survey
GROUP BY species_id
ORDER BY subtotal;
QUIT;
```

4.4. Null values

Using the view table we created in the previous section (`summer_2000`), let's talk about null values. *Missing values* in SQL are identified with the special NULL value. Scroll through our `summer_2000` view. It should be easy to find several records with missing values. How do you think we could filter to find these rows?

To find all records where the species_id is missing, we can use:

```
* keyword IS;
PROC SQL;
SELECT *
FROM sql.survey
WHERE species_id IS NULL;
QUIT;
```

If we wanted to use all the records where `species_id` is NOT null, we would add the `NOT` keyword to our query.

```
* keyword IS NOT;
PROC SQL;
SELECT *
FROM sql.survey
WHERE species_id IS NOT NULL;
QUIT;
```

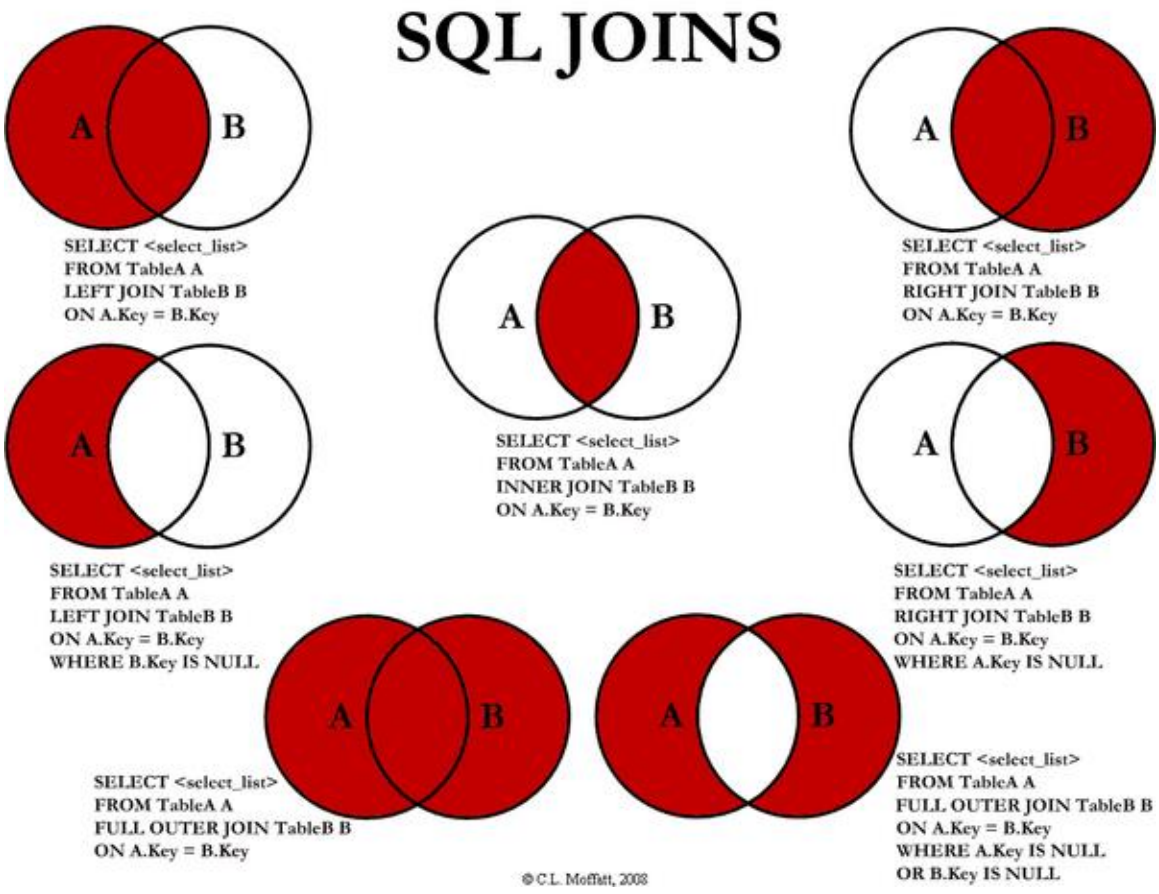
If we restrict our query to the “PE” species, this will be easier to see:

```
* for non-missing values, we use IS/IS NOT or =/!=;
* CAUTION: "=" and "==" both work in SQL, However, "==" does NOT in SAS.;
PROC SQL;
SELECT SUM(weight),
        COUNT(*),
        SUM(weight)/COUNT(*)
FROM sql.survey
WHERE species_id = 'PE';
QUIT;

* for non-missing values, we use IS/IS NOT or =/!=;
* CAUTION: "=" and "==" both work in SQL, However, "==" does NOT in SAS.;
* != works in SQL, but not in SAS. ^= works in SAS;
PROC SQL;
SELECT SUM(weight),
    COUNT(*),
    SUM(weight)/COUNT(*)
FROM sql.survey
WHERE species_id ^= 'PE';
QUIT;
```

5. Joins and aliases

When working with multiple tables in a relational database, we need to use operation JOIN. There are different ways to join tables. The following chart illustrates different JOINs.



5.1. Joins

To combine data from two tables we use the SQL `JOIN` command, which comes after the `FROM` command.

The `JOIN` command on its own will result in a cross product, where each row in first table is paired with each row in the second table. Usually this is not what is desired when combining two tables with data that is related in some way.

For that, we need to tell the computer which columns provide the link between the two tables using the word `ON`. What we want is to join the data with the same species codes.

```
*INNER JOIN;
*Need to use ALIAS to rename/name the data set since
 the files are stored in the SAS permanent libaray;
PROC SQL;
SELECT *
FROM sql.survey AS surveys
JOIN sql.species AS species
ON surveys.species_id = species.species_id;
QUIT;

* we can simply rename the tables as A and B using alias;
```

```
PROC SQL;
SELECT *
FROM sql.survey AS A
JOIN sql.species AS B
ON A.species_id = B.species_id;
QUIT;
```

`ON` is like `WHERE`, it filters things out according to a test condition. We use the `table.colname` format to tell the manager what column in which table we are referring to. The above example illustrates the typical INNER JOIN. The next few examples show how to perform (OUTER) LEFT JOIN, (OUTER)RIGHT JOIN and (OUTER) FULL JOIN.

```
*left join;
PROC SQL;
CREATE TABLE sql.LEFTJOIN AS
SELECT *
FROM sql.survey AS A
LEFT JOIN sql.species AS B
ON A.species_id = B.species_id;
QUIT;

*right join;
PROC SQL;
CREATE TABLE sql.RIGHTJOIN AS
SELECT *
FROM sql.survey AS A
RIGHT JOIN sql.species AS B
ON A.species_id = B.species_id;
QUIT;

*full join;
PROC SQL;
CREATE TABLE sql.FULLTJOIN AS
SELECT *
FROM sql.survey AS A
FULL JOIN sql.species AS B
ON A.species_id = B.species_id;
QUIT;
```

We can select some variables from individual tables then join the two sub tables.

```
* We can select some variables from individual tables
* then join the two sub tables.
* CAUTION: We will NOT select any variables in species table
*  to include in the new table.
PROC SQL;
SELECT A.species_id,
       A.sex,
       AVG(a.hindfoot_length) as mean_foot_length
FROM sql.survey  AS A
JOIN sql.species AS B
ON A.species_id=B.species_id
```

```
WHERE taxa = 'Rodent' AND A.sex IS NOT NULL
GROUP BY A.species_id, A.sex;
QUIT;
```

5.2. Complex Nest Queries

SQL queries help us *ask* specific *questions* which we want to answer about our data. The real skill with SQL is to know how to translate our scientific questions into a sensible SQL query (and subsequently visualize and interpret our results).

What is the percentage of each species in each taxa?

```
* nest queries;
* The SELECT ... FROM in the denominator is self-contained
* It is NOT affected by GROUP BY statement. THe COUNT() function
* returns the total size of the data;
PROC SQL;
SELECT B.taxa,
    100.0*COUNT(*)/(SELECT COUNT(*) FROM sql.survey)  AS Percentage
FROM sql.survey AS A
JOIN sql.species AS B
ON A.species_id = B.species_id
GROUP BY taxa;
QUIT;
```