# Lecture 2 [Working with Datasets]

Gwinyai T.

May 15, 2017

# 2 Working with Datasets

## 2.1 Permanent and Temporary Tables

Launch SAS and navigate to the *Work* folder. It should be empty. *Work* only retains tables for the duration of a SAS session. You may find it convenient to keep some tables permanently. Achieving this is straightforward. In the *Explorer* panel, right click and select new library. Enter the name of your proposed table repository (there is an 8 character limit), check "enable at startup" and, finally, point to a directory on your hard drive in the path box. Exit SAS, relaunch and verify that your newly created library is still there. You can now recreate the students and stocks tables by running the code from the first lecture, and, if you so wish, copy/paste them for access later.

## 2.2 Joining and Splitting Tables

If the reader is starting to develop the impression that a goodly portion of finance work involves data manipulation, that is probably because it is true. Model design is limited in its usefulness without an infrastructure of coherent, accurate observations. After formulating a hypothesis, data are either combined, stripped, or both, until a new collection is produced for statistical analysis and forecasting. SQL has a group of intuitive commands for that exact purpose. The most important ones are discussed herein.

### 2.2.1 Inner Join

The result set of an inner join contains all records from the selected tables that have common keys. Each one of the securities in *_stock01* is associated with a person. Perhaps Alex, Gigi, Jacob et al. are investment analysts whose coverage assignments are LULU, XOM, CBRE etc. Alternatively, the six individuals could be important shareholders in these firms. Whatever the case, there is some connection between *_student01* and *_stock01*. Preparing SQL to handle that connection is done by giving a name and data type to the link variable, then assigning values to facilitate a succesful match.

```
/*  Create a common key on _stock01 */
proc sql;
  alter table _stock01
    add ID int label = 'Identification Number';
  update _stock01
    set ID = monotonic();
quit;
```

The language involved in constructing any type of merge is similar to that which was taught in the previous lecture. First, one instantiates a table using a *create* statement. Second, variables of interest are selected. Finally, any conditional operations are performed. To this point, that last step has consisted of *where* and *group by* clauses performed on the contents of the base table. In this lesson, conditional processing is done on the common keys between the base and its counterpart(s).

```
/*  Inner join  */
proc sql;
  create table _student01stock01
    as select
      a.*,
      permno, comnam, ticker
    from _student01 as a, _stock01 as b
  where a.ID = b.ID;
quit;
```

SQL requires precise designations of which variables are to be pulled and where they can be found. On the subject of where, descriptive names can make for unwieldy code. Table aliases are often used to aid in legibility. Rather than retyping *_student01* and *_stock01* every time something is needed, the letters *a* and *b* serve as stand-ins. Notice that *b's* sole appearance occurs during conditioning. Since *permno*, *comnam* and *ticker* only exist on the second table, there is no ambiguity concerning their origin. Just as aliases tidy up code blocks, wildcards have a similar function. Represented by the * character, wildcards instruct SQL to perform a task on all entities in a given context. For example, the plain English translation of *select a.** is: Select all variables from the location which has the alias *a*.

### 2.2.2 Subsetting Data

The forthcoming examples will feature tables that lack perfect correspondence with one another. More often than not, this will be the case in practice. Were *_student01* expanded to include all students on the class roster, some members might have no meaningful relationship with the elements of the second table, and vice versa. To illustrate such scenarios, identification numbers one through three will be separated from four through six. A single call of *proc SQL* creates four tables for further analysis.

```
/*  Subsetting data */
proc sql;
  create table _student01FirstHalf
    as select
      *
    from _student01 where ID < 4;
  create table _student01SecondHalf
    as select
      *
    from _student01 where ID > 3;
  create table _stock01FirstHalf
    as select
      *
    from _stock01 where ID < 4;
  create table _stock01SecondHalf
    as select
      *
    from _stock01 where ID > 3;
quit;
```

### 2.2.3 Left Outer Join

The result set of a left outer join contains all records from the base table, and the records with matching keys from its counterpart. Using the data at hand, an application that utilized the names of the six students, but only the equity preferences of the first three would, as a first step, require a left outer join. The conditioning syntax of outer joins acknowledges that mismatches are inherent. Instead of *where* (which speaks to all locations having strict equality), the term *on* is used. Verify that your output consists of six people and three stocks.

```
/*  Left outer join */
proc sql;
  create table _left01
    as select
      a.*,
      permno, comnam, ticker
    from _student01 as a left join _stock01FirstHalf as b
      on a.ID = b.ID;
quit;
```

### 2.2.4 Right Outer Join

The result set of a right outer join contains all records from the companion table, and the records with matching keys from the base. Changing the direction of the *join* command flips the importance of people with equities. Verify that your output consists of the first three people and all six stocks.

```
/*  Right outer join  */
proc sql;
  create table _right01
    as select
      a.*,
      permno, comnam, ticker
    from _student01FirstHalf as a right join _stock01 as b
      on a.ID = b.ID;
quit;
```

### 2.2.5 Full Join

The result set of a full join contains all records from both tables, irrespective of whether a match exists. Try running this program:

```
/*  Full join */
proc sql;
  create table _full01
    as select
      a.*,
      permno, comnam, ticker
    from _student01 as a full join _stock01 as b
      on a.firstName = b.comnam;
quit;
```

Justifying this query from a theoretical standpoint is a tad risible. Maybe the programmer suspects the students are so self-absorbed that they only supply analyst opinions, or investment capital for companies bearing their own names. Unfortunately, no such cases exist. SQL returns the union of *_student01* and *_stock01*. Wherever a field does not match a record, a missing value is generated. Missing numerical data are represented by a period, whilst missing string data do not appear on screen at all. SAS code for a blank entry is *" "*.

Had the program been run on common fields, the nature of the merge would be apples to apples. Although nothing insightful is gained from the apples to oranges table, building it demonstrates an important programming point. Imagine you are a farmer who delivers perishables to market everyday. This morning you have a tray with six apples on your left, another with six oranges to your right, a roll of duct tape, a labeling gun, and a pair of scissors. Both trays hold up to twelve items, but the storage receptacles are crafted to only accomodate the specific type of fruit. What would you do? A rational farmer might inspect both trays, snip them in half, tape the halves containing fruit together (for ease of transportation), keep the empties for future use, and label the deliverables appropriately. How does SQL, anthropomorphized as a farmer, approach this problem?

SQL selects an apple and compares it to the contents of the first bay she finds on the right. If empty, she cuts the bay off, carves out a replacement from the left tray, tapes that in place, and labels the newly inserted fruit, "apple; not orange." On the other hand, if a right side slot is occupied, she labels its contents as "orange; not apple," and resumes the search for an empty bay. This process is repeated until all the apples from the left are transferred to the right. From the perspective of a buyer at the market, her end product is the same as yours. Unlike with you, and coding routines that "think" like you (SAS's *merge* command), the order of the elements in the tray does not matter to SQL. This has consequences for execution speeds, and is the focus of the lecture *Log Times and Loops.* For now, reconduct the thought experiment, but with four hundred apples and six hundred oranges arrayed haphazardly, and a buyer whom only cares about when produce arrives at market.

The labeling gun in the analogy is also instructive for full joins on elements that match. Envision a scenario with an arbitrary (but known) number of left side apples, and an unknown number of right side apples. Suppose SQL starts with apple number three hundred and fifty seven in her hand, and finds an empty bay on the right. That apple will be inserted, labeled "apple 357," and the delicious spread will be one step closer to delivery. In the second run, SQL holds the fourth apple from the left, and on the right, finds a slot occupied by the ninetieth apple. It stands to reason that the tray must have a fourth apple of its own, but this does not cause a conflict yet. SQL therefore proceeds as normal. When the item already bearing the label "apple 4" is discovered, a duplicate is discarded. Unless her unstated goal is to open a cider mill, SQL the farmer needs to be advised on how to handle ties.

Recall that primary keys identify distinct, indivisible entities. The problem in the orchard is that "apple 4 - left" is indistinguishable from "apple 4 - right." Whether explicitly stated in the select clause, or left up to the computer to intuit, *coalesce()* logic dictates that only the first non-null entry in a series is kept. To avoid throwing away potentially useul data (or apples), keys in a full join should not overlap.

```
/*  Apples to apples */
proc sql;
  create table _full01
    as select
      coalesce(a.ID, b.ID) as ID,
      coalesce(a.firstName, b.firstName) as firstName,
      coalesce(a.lastName, b.lastName) as lastName,
      coalesce(a.calendarDate, b.calendarDate)
        as calendarDate format yymmdd10.
    from _student01FirstHalf as a
      full join _student01SecondHalf as b
        on a.ID = b.ID;
quit;
```

### 2.2.6 Concatenate

A visual inspection confirms that *_full01* is indeed a replica of *_student01*. With more involved comparisons, *proc compare* would be a preferable method for the task. Stacking tables in a database is called vertical concatenation. Like all the preceding types of joins, the syntax for concatenation is borrowed from set theory. Picture a Venn diagram that shows the students assigned to their respective halves (A and B). The set that includes all students is $A \cup B$, and can be visualized as two overlapping circles shaded in the same color. The shaded area accounts for all the elements in A and not in B, all the elements in B and not in A, and all the elements common to both. SQL abbreviates this description as *outer union*. Notice that $A \cup B$ is relevant to both the row (horizontal concatenation) and column (vertical concatenation) dimensions. Distinguishing one type of query from the other is done with the *corr* switch. Telling the engine that there is a correlation between fields in A and fields in B shapes the final table into the same dimensions as *_student01* and, of course, *_full01*. With your knowledge of wildcards, in conjunction with the information in this paragraph, you should be able to create a clone of *student01* using fewer commands than before.

```
/*  Concatenate */
proc sql;
  create table _concatenate01
    as select
      * from _student01FirstHalf
    outer union corr
      select
      * from _student01SecondHalf;
quit;

/*  Check the Results Viewer for comparative output */
proc compare
  base = _student01 compare = _concatenate01;
quit;

/*  Fin!  */
```

Verify your results, save your code and exit SAS.