Using Enterprise Guide for Table Joins and Sets

J. Warren Schlechte, Texas Parks and Wildlife Department, Mt. Home, TX
John B. Taylor, Texas Parks and Wildlife Department, Austin, TX

Abstract

Enterprise Guide (EG) provides a painless way to learn how to implement SQL joins. Basic SQL joins (inner and outer) can be created using the Filter/Query tool. Enterprise Guide allows the user to see the coding used for these joins, thereby giving the user the ability to learn SQL coding in a point-and-click environment. In addition to joins, SQL can be used for set operations. However, set operations require the user to code in SQL. By copying the SQL code from EG and slightly modifying it using keyword syntax, the user has a tremendous set of tools for combining and manipulating data sets.

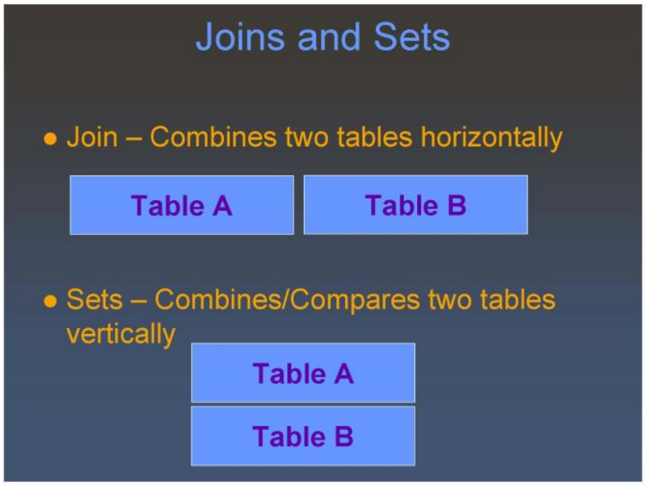
Introduction

In most organizations, data for any given project is likely to be scattered across several tables (or datasets). Data can reside on different servers, and be stored using different software. Proc SQL provides a tool for grabbing the data you need from these various tables. Typically, to get the data you need from the tables that already exist, one links the different data tables through key fields (or columns). By matching values in key fields in one table with key fields in another tables, the user can create a new table (or view) with only the pertinent information.

Using joins via Proc SQL, the user can subset and combine tables to create a new table that is unique and provides the information needed to answer their questions. In addition, SQL allows the user to produce summaries of the data quite easily. This is useful when doing data exploration. SAS EG allows the user to create these links in a point and click environment.

At other times, joins, because they are based on combining data across key fields, are not what are desired. Instead, what you may want to do is to either combine all data from two data sources, update data based on one source, or pull only unique data from a table. These operations are set operations. For those of you familiar with Venn Diagrams and set operations in mathematics, these operations are known as union, intersection, and complement.

The intersection of A and B is defined as $A \cap B = \{ x \mid x \text{ in A and x in B} \}$.
The union of A and B is defined as $A \cup B = \{ x \mid x \text{ in A or x in B} \}$.
The complement of A and B is defined as $A \backslash B = \{ x \mid x \text{ in A and x not in B} \}$.



Joins and Sets

- Join – Combines two tables horizontally

Table A Table B

- Sets – Combines/Compares two tables vertically

Table A

Table B

Figure 1: Joins and Sets

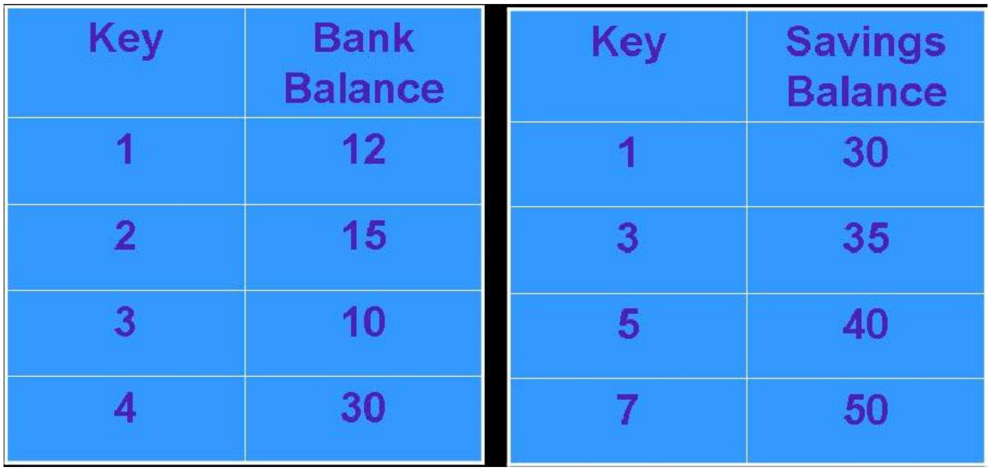
Set operations can also be done using SQL. EG is much better working with joins than with sets, but if you start to use EG and understand what is going on, you can use that code and extend it by adding your own coding. Because having both joins and sets are useful for many data manipulations, and because both can be coded in Proc SQL, it makes sense to use the point-and-click environment of EG to learn how to do joins, then extend your abilities using the code provided by EG, along with some SAS keywords to cover set operations.

For those of you used to working in the data step processes, joins are equivalent to a “merge” or “match merge”, whereas the sets are equivalent to the “set” statements, or are sometimes associated with dataset updating.

We’ll start with Joins using some simple examples.

Data

Here are two datasets that an organization may have. In the first table, we see people with unique identifiers (the “key” field) and their bank balance. In the second table, we see people, again with unique identifiers, and their savings balance. Notice that persons 1 and 3 have both a bank and savings account, 2 and 4 have bank accounts only, and 5 and 7 have savings accounts only.



Key	Bank Balance
1	12
2	15
3	10
4	30

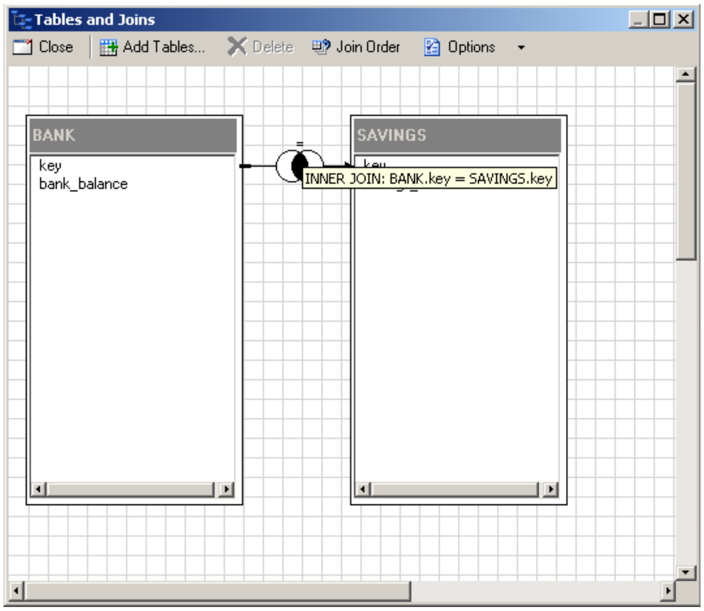
Key	Savings Balance
1	30
3	35
5	40
7	50

Figure 2: Tables showing bank account and savings account information

Cartesian Product

All inner joins begin with a Cartesian product of your tables. In a Cartesian product all rows of your first table are combined with all rows of the second table. Thus, for two tables with 4 rows each, the resulting table is a table with 16 rows. Let's use EG and see how we get the basic Cartesian product for these data. (I'm going to assume you've already used EG, and that this will just extend your abilities. If not, there are several good tutorials available for EG online. I suggest you start with those, along with "The Little SAS Book for Enterprise Guide" by Slaughter and Delwiche to help you get started.)

To begin, assume we have two tables, one named "Bank" and one named "Savings". We select one table, let's say "Bank", then use the Filter/Query tool (Data -> Filter and Query). Click on "Add Tables", and add "Savings" from this project. Select all the variables (shift-left-click) and drag them into the Select Data field. Notice the variable "key" from "Savings" has been renamed "key1". Now click on "Join". Notice by default, SAS assumes that you want an inner join based on the field "key".



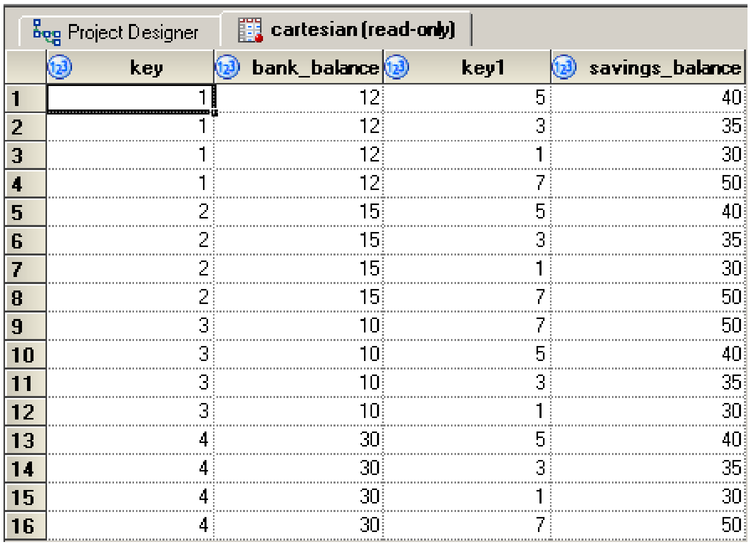
INNER JOIN: BANK.key = SAVINGS.key

Figure 3. Inner Join default

For the moment, we'll ignore what an "Inner Join" means, and create our Cartesian product. To do so right-click the join symbol and hit "Delete Join". You should now have the two tables, with no arrow linking them. Close the Join pop-up and respond "Yes" to the pop-up notifying you of performance issues.

We'll not Filter the data yet, so select the "Sort Data" tab. Say we want to sort by both "key" fields. Drag "key" from the Bank table over, then try to drag "key" from Savings. Notice, this operation cannot be done because whereas in the Select Data tab, "key" was renamed to "key1", such is not the case here. So, let's sort solely by "key", and later I'll show you how to also sort by the second key field as well.

The result of our first join is the Cartesian product shown below. Notice we have 16 rows, and that each "key" field from the "Bank" table is combined with each "key1" field from the "Savings" table.



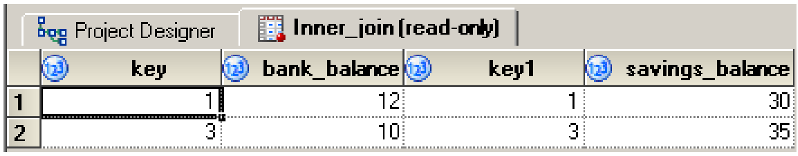
	key	bank_balance	key1	savings_balance
1	1	12	5	40
2	1	12	3	35
3	1	12	1	30
4	1	12	7	50
5	2	15	5	40
6	2	15	3	35
7	2	15	1	30
8	2	15	7	50
9	3	10	7	50
10	3	10	5	40
11	3	10	3	35
12	3	10	1	30
13	4	30	5	40
14	4	30	3	35
15	4	30	1	30
16	4	30	7	50

Figure 4. Cartesian Product of "Bank" and "Savings"

Inner Joins

Inner joins return all rows from one table that have matching rows in a second table (Note: More than two tables can be joined. We are limiting our examples to two tables to keep things simple). For our data, only the “key” field makes sense to use for an inner join. So, do as before, selecting “Bank”, adding “Savings” (Figure 2), then select Join. Notice the default is an inner join on our “key” field, which is what we want (Figure 3). (If we wanted to change which fields we were interested in joining, it is as easy as clicking the variable in the first table and dragging it to the variable in the second table.)

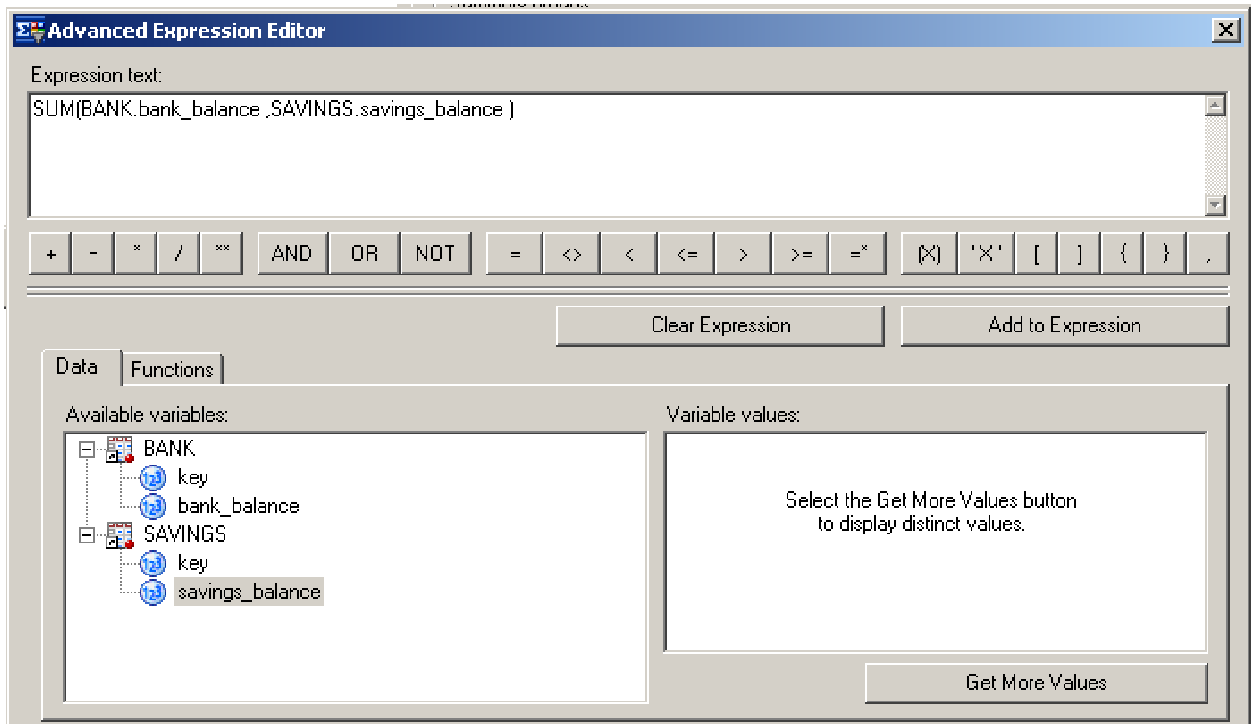
The resulting table (Figure 5) shows what is expected; two rows, those with “key” fields of “1”and “3”.



	key	bank_balance	key1	savings_balance
1	1	12	1	30
2	3	10	3	35

Figure 5. Inner join table

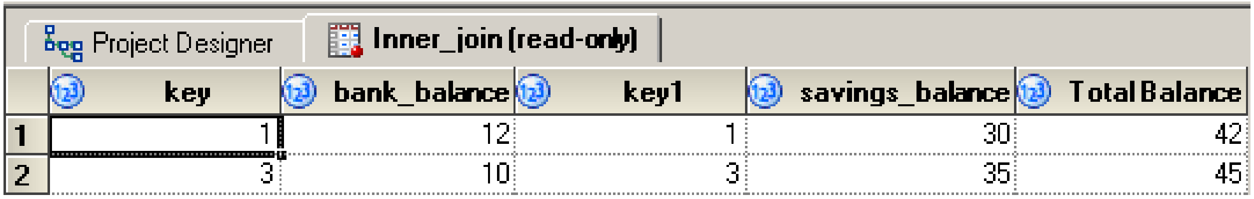
An extension of this simple example is to add a summary column, say the total balance of the bank and savings. Open the query (by double-clicking on it in the Project Designer) and select “Computed Columns”. Select “New”, then “Build Expression”. To compute the total, insert the cursor in the “Expression Text” box and type in “SUM(“, double left-click “bank_balance”, add a comma, double left-click “savings_balance”, and add a right parenthesis (Figure 6). You can rename the default name “Calculated1” by selecting the name, then selecting “Rename”.



Expression text:

```
SUM(BANK.bank_balance ,SAVINGS.savings_balance )
```

Figure 6. Creating the sum as a computed column



	key	bank_balance	key1	savings_balance	Total Balance
1	1	12	1	30	42
2	3	10	3	35	45

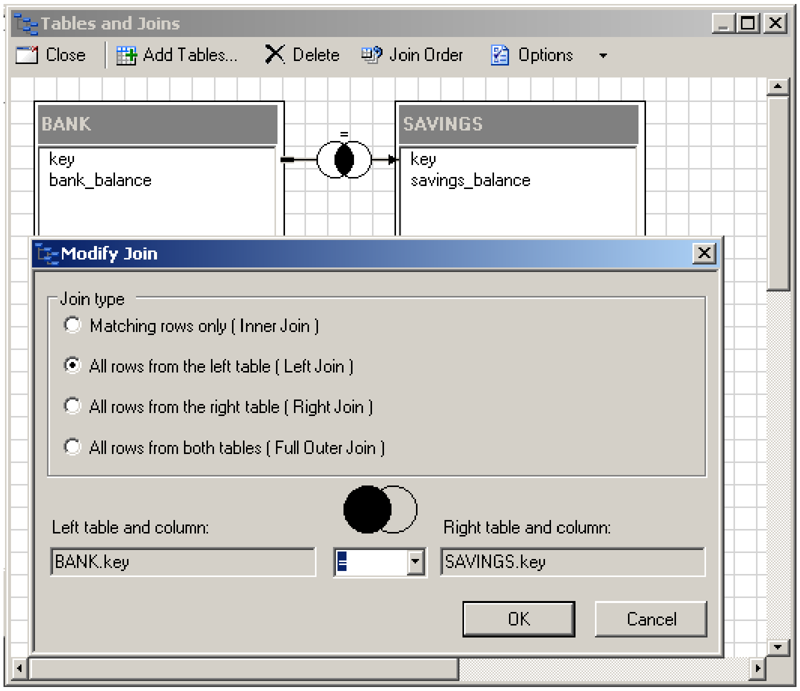
Figure 7. Computed column result

Outer Joins

Previously, we showed where a join links rows in one table with rows in another. Occasionally, you may want to include rows from one or both tables that have no related rows. This type of operation is an outer join.

Left Join

A Left Outer Join takes all rows from the first table (on the left) and includes matching rows from the second table (on the right). To do a left outer join, go through the steps as before, but when you get to the stage where the default join is the inner join, right-click on the join and select "Modify Join". This then allows you to change the inner join to a Left Join, Right Join, or Full Outer Join. Notice the small diagram in the bottom middle reinforces how this join will work. The result will be all those people with bank accounts, and will show, of those, what is their savings balance. We added a computed column as well to show our customers total balance.



Join type

○ Matching rows only (Inner Join)

● All rows from the left table (Left Join)

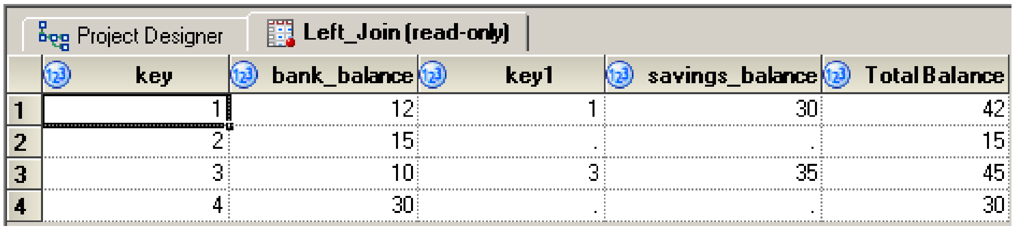
○ All rows from the right table (Right Join)

○ All rows from both tables (Full Outer Join)

Left table and column: BANK.key

=

Right table and column: SAVINGS.key



	key	bank_balance	key1	savings_balance	Total Balance
1	1	12	1	30	42
2	2	15	.	.	15
3	3	10	3	35	45
4	4	30	.	.	30

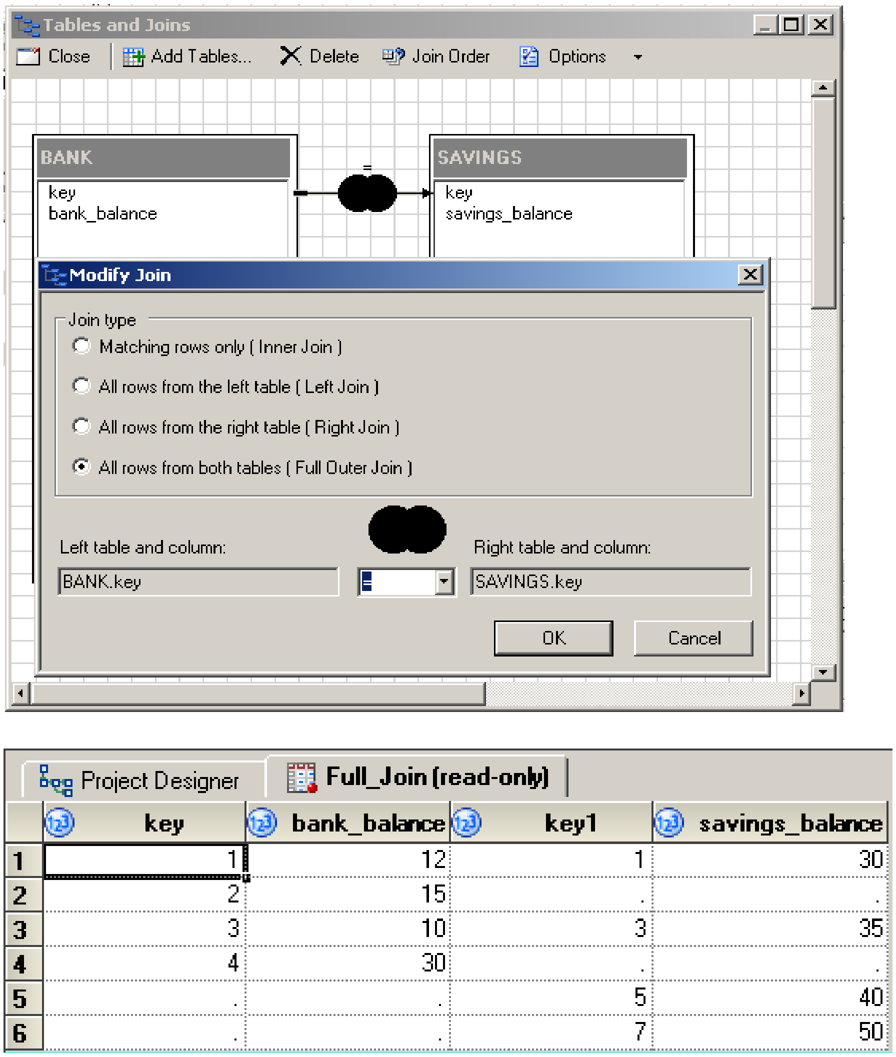
Figure 8. Left Join and resulting data set

Right Join

A Right Outer Join works very similarly. It takes all rows from the second table (on the right) and includes matching rows from the first table (on the left). It is obvious that order is important. If the user reverses the order the tables are selected, they will also need to reverse the type of outer join.

Full Outer Join

A Full Outer Join takes all rows from the first table and all rows from the second table. Initially, this may sound like a Cartesian product, but it is not. Instead of matching each row in the first table with each row in the second table, matching rows are linked, but unmatching rows are also included.



Join type

○ Matching rows only (Inner Join)

○ All rows from the left table (Left Join)

○ All rows from the right table (Right Join)

● All rows from both tables (Full Outer Join)

Left table and column: BANK.key

Right table and column: SAVINGS.key

	key	bank_balance	key1	savings_balance
1	1	12	1	30
2	2	15	.	.
3	3	10	3	35
4	4	30	.	.
5	.	.	5	40
6	.	.	7	50

Figure 9. Full Outer Join and resulting dataset

While all the pertinent data are here, this default arrangement is odd because your new dataset has two “key” fields. What you likely want is a single “key” field, with missing data for “savings_balance” for those with just bank accounts, and missing data for “bank_balance” for those customers with just savings accounts. You might be tempted to think this could be achieved in EG by only including “key” from “Bank”. Unfortunately, that does not work, as the “key” field for customers 5 and 7 are coded as missing values. We’ll resolve this issue later when we learn how to use the COALESCE keyword.

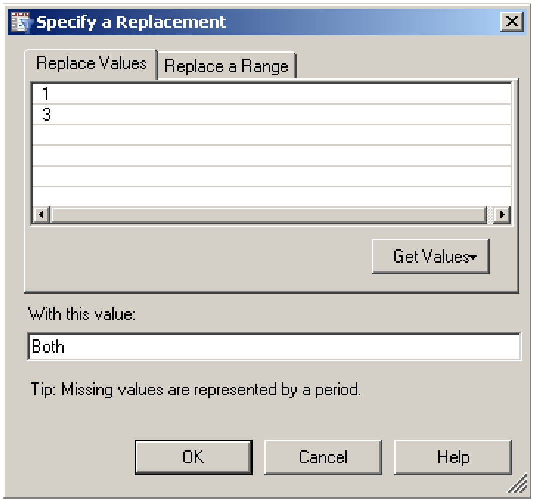
Extending Inner and Outer Joins - I

Let’s use what we’ve learned to answer a simple question, “Which of our customers gives us the most money; those with bank accounts only, those with savings accounts only, or those with both?” To do so we will make use of three queries to the original data sets, use the computed columns options, and filter the data.

Both Bank Accounts and Savings Accounts

Customers with both bank accounts and savings accounts refers to the dataset created using the inner join. To calculate the average amount these users have in our location, we need to summarize the data. One way to do this is the following:

Select the Inner Join dataset. Create two new computed columns. Name the first “Avg”, and create it as AVG(Total Balance) using the “Build an Expression” as before.
Create the second computed column using the “Recode Columns” option. What we’re trying to do is label this average with the string “Both”. Select any one of the existing variables, for example “key”. Name the new variable by typing “Type” in the “New Column Name” field. Left-click on the “Add” button, then on the “Replace Values” tab, click on get values. Select both “1” and “3”. In the “With This Value” field, enter the string “Both”. Make the new column a character variable using the radio buttons at the bottom of the pop-up, and left-click “OK”.



1
3

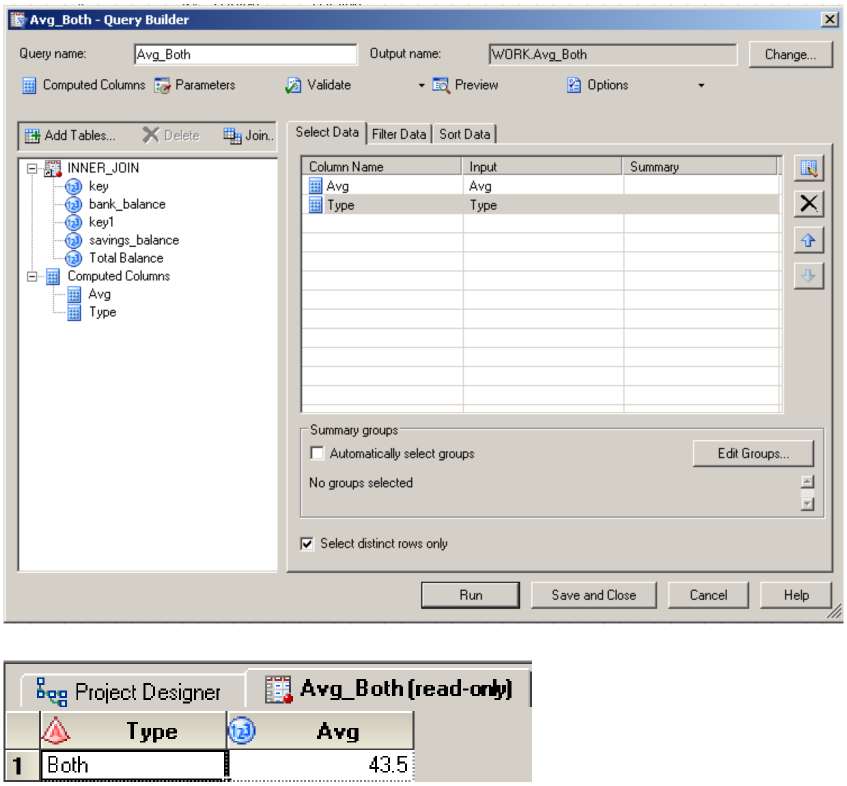
With this value:

Both

Tip: Missing values are represented by a period.

Figure 10. Recoding the "key" field to a field containing the string "Both"

Select only the new variables “Avg” and “Type” for output, and click the “Select distinct rows only” checkbox at the bottom of the window. Our dataset “Inner Join” has two rows, one for customer “1” and a second for customer “3”. However, we want the average total, which is a summary over both customers. If we run the query without selecting distinct rows only, then we would get two rows of output, both with the average amount in the accounts. By using the “Select distinct rows only” checkbox, we only output a single row.



Avg_Both - Query Builder

Query name: Avg_Both

Output name: WORK.Avg_Both

- INNER_JOIN
  - key
  - bank_balance
  - key1
  - savings_balance
  - Total Balance
- Computed Columns
  - Avg
  - Type

Column Name	Input	Summary
Avg	Avg	
Type	Type	

Summary groups

☐ Automatically select groups

No groups selected

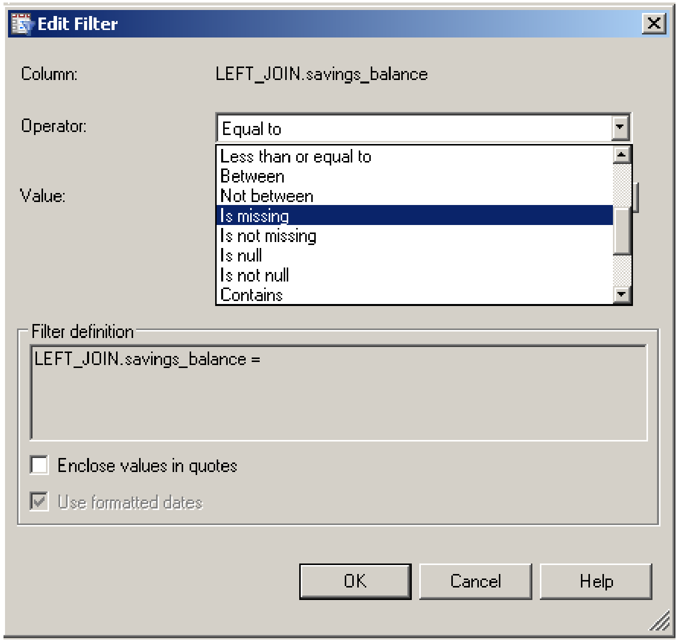
☑ Select distinct rows only

	Type	Avg
1	Both	43.5

Figure 11. Creating a new table to reflect deposits from customers with both bank and savings accounts.

Only Bank Accounts

Customers with only bank accounts can be gotten from the dataset created using the left join, removing those who also have savings accounts. As with the previous dataset, you will need to calculate the average account balance and create the "Type" variable – this time the value of "Type" should be "Bank", not "Both" to reflect these are customers with only bank accounts. To remove those with saving accounts, we will use the Filter tab. Select the "savings_balance" variable, and use the "Is Missing" option to remove these from our table.



Edit Filter

Column: LEFT_JOIN.savings_balance

Operator: Equal to

Value:

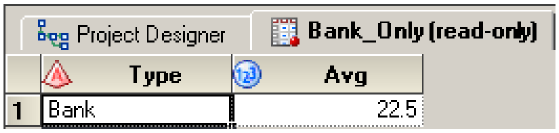
Filter definition

LEFT_JOIN.savings_balance =

☐ Enclose values in quotes

☑ Use formatted dates

Figure 12. Filtering to remove unwanted rows.

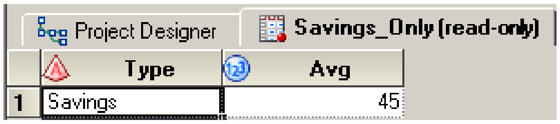


	Type	Avg
1	Bank	22.5

Figure 13. Table of deposits for customers with only bank accounts.

Only Savings Accounts

Customers with only bank accounts can be gotten from the dataset created using the right join, removing those who also have bank accounts. As with the previous dataset, you will need to calculate the average account balance and create the "Type" variable – this time the value of "Type" should be "Savings". To remove those with bank accounts, we again will use the Filter tab. Select the "bank_balance" variable, and use the "Is Missing" option.



	Type	Avg
1	Savings	45

Figure 14. Table of deposits for customers with only savings accounts.

Looking at the three tables, we see people with only savings accounts have an average of 45 units, people with both bank and savings accounts have an average of 42.5 units, and people with only bank accounts have an average of 22.5 units. This suggests that people with savings accounts deposit the most money into our bank.

Extending Inner and Outer Joins - II

The previous operations used EG's point-and-click interface. However, earlier we encountered two problems where the point-and-click approach could not get us the output we wanted. Fortunately, in EG, the underlying SAS code is always available and with it, and some keywords, we will be able to accomplish much more.

Here is the basic SQL structure:

```
Proc SQL;
   Create Table table_name as
         Select variable(s)
         From table(s)
         Where condition(s)
         Group by variable(s)
         Having condition(s)
         Order by variable(s)
    ;
Quit;
```

Some quick notes about SQL code. First, order of the keywords is important, so Select comes before From, etc. (a mnemonic to remember the ordering is helpful here). Second, not all keywords need to be used for each query (you'll see how this works soon). Third, unlike other SAS code, variables are segregated using commas. Fourth, the only semi-colons used are at the end of the Proc SQL line, and after the full set of keywords.

The underlying SQL code is captured by right-clicking on the Query, and selecting "Open Last Submitted Code". Take the time now to examine the SQL code behind each Query to see the differences between the Cartesian product, the Inner Join and the Outer Joins.

The first example where we got stuck using only point-and-click EG was on the Cartesian product. Recall how we could not sort on both "key" fields from both datasets because in the "Sort By" tab, EG would not distinguish between "key" in the "Bank" table and "key" in the "Savings" table. If we pull up the code that was used, we see:

```
PROC SQL;
 CREATE TABLE SCSUG.Cartesian AS
 SELECT BANK.key,
       BANK.bank_balance,
       SAVINGS.key AS key1,
       SAVINGS.savings_balance
 FROM SCSUG.BANK AS BANK,
      SCSUG.SAVINGS AS SAVINGS
 ORDER BY BANK.key;
QUIT;
```

In the `SELECT` portion of the SQL code, notice that SAVINGS.key is aliased (using the `AS` keyword) as key1 (we saw this in the Select Data tab). Unfortunately, this alias did not appear in the "Sort Data" tab. By using the code, we can get EG to do what we want. A simple change in `ORDER BY` the line to

```
ORDER BY BANK.key, key1;
```

and we get the output we desire.

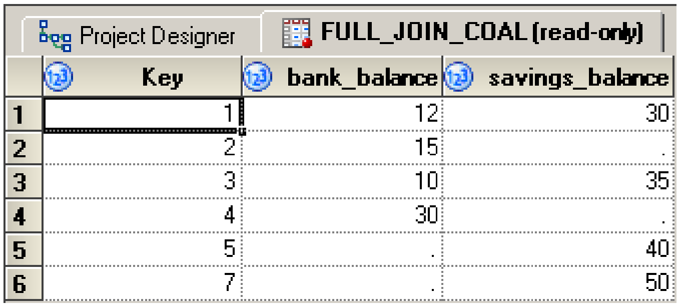
Similarly, in our Full Outer Join example, EG forced us to have two "key" fields, but we only wanted a single "key" field. The SQL code for the Full Outer Join is

```
PROC SQL;
 CREATE TABLE SCSUG.FULL_JOIN AS
 SELECT BANK.key,
       BANK.bank_balance,
       SAVINGS.key AS key1,
       SAVINGS.savings_balance
 FROM SCSUG.BANK AS BANK
        FULL JOIN SCSUG.SAVINGS AS SAVINGS ON (BANK.key = SAVINGS.key);
QUIT;
```

To have a single "key" field one must use the keyword: COALESCE

```
PROC SQL;
 CREATE TABLE SCSUG.FULL_JOIN_COAL AS
 SELECT COALESCE (BANK.key,SAVINGS.key) as Key label="Key",
       BANK.bank_balance,
       SAVINGS.savings_balance
 FROM SCSUG.BANK AS BANK
      FULL JOIN
      SCSUG.SAVINGS AS SAVINGS ON (BANK.key = SAVINGS.key);
QUIT;
```

The COALESCE keyword tells SQL to create a single variable "Key" from Bank.key and Savings.key, replacing the missing values from each with a non-missing value.



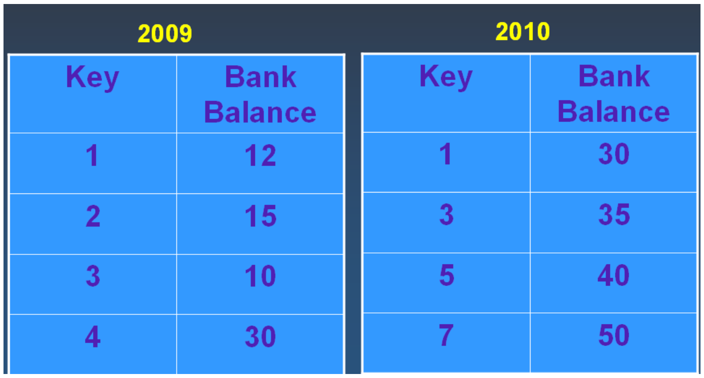
	Key	bank_balance	savings_balance
1	1	12	30
2	2	15	.
3	3	10	35
4	4	30	.
5	5	.	40
6	7	.	50

Figure 15. Full outer join with a single "Key" field.

Notice the difference between the newly created dataset (Figure 15) and Figure 9 on page 7.

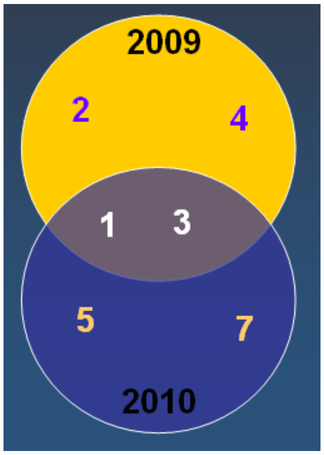
Set Operations

As the previous two examples showed, small additions to existing EG code is a good way to extend SQL to do what you want it to do. In the latter example, it was important to know a specific keyword to accomplish the task. The same is true for Set Operations. The four keywords you need to know are INTERSECT, UNION, EXCEPT, and OUTER UNION.



2009		2010	
Key	Bank Balance	Key	Bank Balance
1	12	1	30
2	15	3	35
3	10	5	40
4	30	7	50

Figure 16. Datasets used to illustrate Set Operations.



To illustrate how set operations work, I will use these two new datasets (Figure 16). Assume that in 2009, we had four customers, and in 2010, we again had four customers. Two customers were retained across both years ("1" and "3"), two were lost in 2010 ("2" and "4"), and two were gained in 2010 ("5" and "7"). This is also illustrated in the Venn Diagram.

To find those customers retained in both years is the same as asking for the intersection of the two customer fields. This is done by:

```
PROC SQL;
CREATE TABLE scsug.kept_in_2010 AS
SELECT key
FROM scsug.bank_2010
 INTERSECT
SELECT key
FROM scsug.bank_2009
;
QUIT;
```

To find all customers who came into our bank regardless of year is the same as asking for the union of the two customer field. This is done by:

```
PROC SQL;
CREATE TABLE scsug.all AS
SELECT key
FROM scsug.bank_2010
 UNION
SELECT key
FROM scsug.bank_2009
;
QUIT;
```

To find all customers who were unique to either 2009 or 2010, you would use the EXCEPT keyword. We'll leave this for you to do as practice.

A SAS extension to SQL is the OUTER UNION, which works like the SET keyword in a DATA statement. Variables that are the same in both tables are appended. Variables that are unique to each table are retained, but the values are set to MISSING in the rows that come for the table without the unique variable. If we wanted to join the three datasets we created in "Extending Inner and Outer Joins – I", we would use an OUTER UNION, like so:

```
PROC SQL;
CREATE TABLE SCSUG.All_Avg AS
SELECT * FROM SCSUG.AVG_BOTH
 OUTER UNION CORR
SELECT * FROM SCSUG.BANK_ONLY
 OUTER UNION CORR
SELECT * FROM SCSUG.SAVINGS_ONLY
;
Quit;
```

In this example, you see use of the wildcard symbol "*", which means "ALL". So, `SELECT *` means "Bring all the variables from this dataset into the new dataset". You also see the keyword "CORR", which means, if these columns have the same name, treat them as the same variable.

CONCLUSION

SQL is a powerful tool, and using EG, you can begin to learn and implement SQL queries, joins and set operations. This tool provides a way to combine, subset, or add summary data to your tables. Using EG is easy, and provides an intuitive visual to help you learn and understand what SQL joins are doing. Because it also provides you with the code it is using, you can use the code for learning and for extending the SQL code.

SAS's SQL procedure follows most of the American National Standards Institute (ANSI) guidelines, although it is not fully compliant. This means most of the SQL you learn can be used on other systems. The simplicity and flexibility of performing joins with the SQL procedure makes it an especially useful tool for data gathering and manipulations.

REFERENCES

The Little SAS Book for Enterprise Guide 4.2. 2010. Susan J. Slaughter and Lora D. Delwiche.
SAS Publishing, Cary, NC.