

Unanticipated Results: Uncovering Five Unexpected Outcomes in PROC SQL Joins

Kiran Venna

Abstract

This paper explores five scenarios where unexpected results may occur when using PROC SQL joins. The first scenario investigates the impact of missing values in datasets used in a join. The second scenario examines how the order of multiple tables in a join can affect the results. The third scenario focuses on unexpected outcomes in a LEFT JOIN when a WHERE clause is used instead of an AND condition. The fourth scenario addresses the issue of joining datasets without considering duplicates from the joining datasets. Lastly, the fifth scenario explores the potential for unexpected results when using joins based on the INDEX function or the LIKE operator.

Introduction

Joins are essential in SQL, enabling the merging of data from different but related datasets into a single unified set. This merging improves data comprehension, which is crucial for analysis, reporting, and modeling.

This paper explores five scenarios where SQL joins can produce unexpected results. Each scenario is illustrated with examples and corresponding results, along with strategies to avoid these errors through code adjustments or general best practices.

Impact of missing values in datasets used in join

To understand the impact of missing values in joins, let's examine two datasets where the join variables contain missing values. In the example below, the ID variable, which serves as the joining variable, has missing values in both datasets.

```

data one;
    input id name $;
    datalines;
1 Smith
. John
2 Sam
;
run;

data two;
    input id age;
    datalines;
1 30
. 40
2 45
;
run;

```

We can join the aforementioned datasets using the following code:

```

proc sql;
select      a.*
            ,age
from        one a
inner join  two b
on          a.id=b.id;
quit;

```

To understand the outcome of joining these tables, we can examine the results shown below:

id	name	age
1	Smith	30
.	John	40
2	Sam	45

The results indicate that a missing value in the joining column of dataset one is treated as equal to a missing value in the joining column of dataset two. This behavior is unusual because, in most databases, missing values are not considered equal and are typically ignored during comparisons.

Treating missing values as equal can cause significant issues, such as many-to-many join. Let us examine this problem with the examples below, where both datasets being joined have multiple missing values in their joining columns.

```

data one_1;
    input id name $;
    datalines;
1 Smith
. John
. Megan
2 Sam
;
run;

```

```

data two_2;
    input id age;
    datalines;
1 30
. 40
. 50
2 45
;
run;

```

If we want to join these datasets based on the ID variable, the code will be as shown below.

```

proc sql;
select a.*
       ,age
from one_1 a
inner join two_2 b
on a.id=b.id;
quit;

```

From the results below, we can see that we have many-to-many join, where each missing value in one table joins with each missing value in the other. This not only creates erroneous results but also has the potential to generate exceptionally large datasets.

id	name	age
1	Smith	30
.	John	40
.	John	50
.	Megan	40
.	Megan	50
2	Sam	45

To solve this problem, unless the business specifically demands these results (which is rare), the best approach is to explicitly specify in our code that joins with missing values should be excluded, as shown below.

```
proc sql;
select      a.*
           ,age
from        one_1 a
inner join  two_2 b
on          a.id=b.id
and         a.id is not null
and         b.id is not null;
quit;
```

The above code prevents any null values from dataset one_1 from joining with null values from the dataset two_2 dataset and produces results as shown below.

id	name	age
1	Smith	30
2	Sam	45

How the order of joins within multiple tables can affect the results.

This scenario is applicable when more than two tables are used in joins. Let us try to understand this through the sample datasets shown below.

```
data test1;          data test2;          data test3;
  input id sal;      input id age;      input id bank_amount;
  datalines;        datalines;        datalines;
1 200
3 300
7 400
;
run;                1 35
                    3 90
                    5 29
;
run;                1 10000
                    2 40000
                    7 50000
;
run;
```

To analyze the impact of different join orders, we use Test1, Test2, and Test3. Figure 1 illustrates the results of a LEFT JOIN between Test1 and Test2, followed by an INNER JOIN with Test3, and vice versa. The results are ordered by ID for clarity.

Test1 left join test2 inner join test3

```
proc sql;
  select      a.id,
              a.sal,
              b.age,
              c.bank_amount
  from        test1 a
  left join   test2 b
  on          a.id = b.id
  inner join   test3 c
  on          a.id= c.id
  order by a.id;
quit;
```



id	sal	age	bank_amount
1	200	35	10000
7	400	.	50000

Test1 left join test3 inner join test2

```
proc sql;
  select      a.id,
              a.sal,
              c.age,
              b.bank_amount
  from        test1 a
  left join   test3 b
  on          a.id = b.id
  inner join   test2 c
  on          a.id= c.id
  order by a.id;
quit;
```



id	sal	age	bank_amount
1	200	35	10000
3	300	90	.

Figure 1: order of tables within multi-table join can affect the results.

To understand the differences in results when working with multiple datasets (e.g., three tables), it's important to know that query execution typically follows this pattern: first, two tables are joined based on a specified condition, creating a temporary result set. This temporary set is then joined with the next table based on another condition.

In Scenario 1, Test1 is LEFT JOINed with Test2, resulting in a temporary result set containing all records from Test1 and any matching records from Test2 based on the join column (ID). This temporary result set is then INNER JOINed with Test3. Only records where the join condition between the temporary result set and Test3 is met (highlighted in yellow in Figure 2) will be included in the final output. This process is similar in other scenario, with the main difference being the order of the tables. The impact on the final results is illustrated at each step.

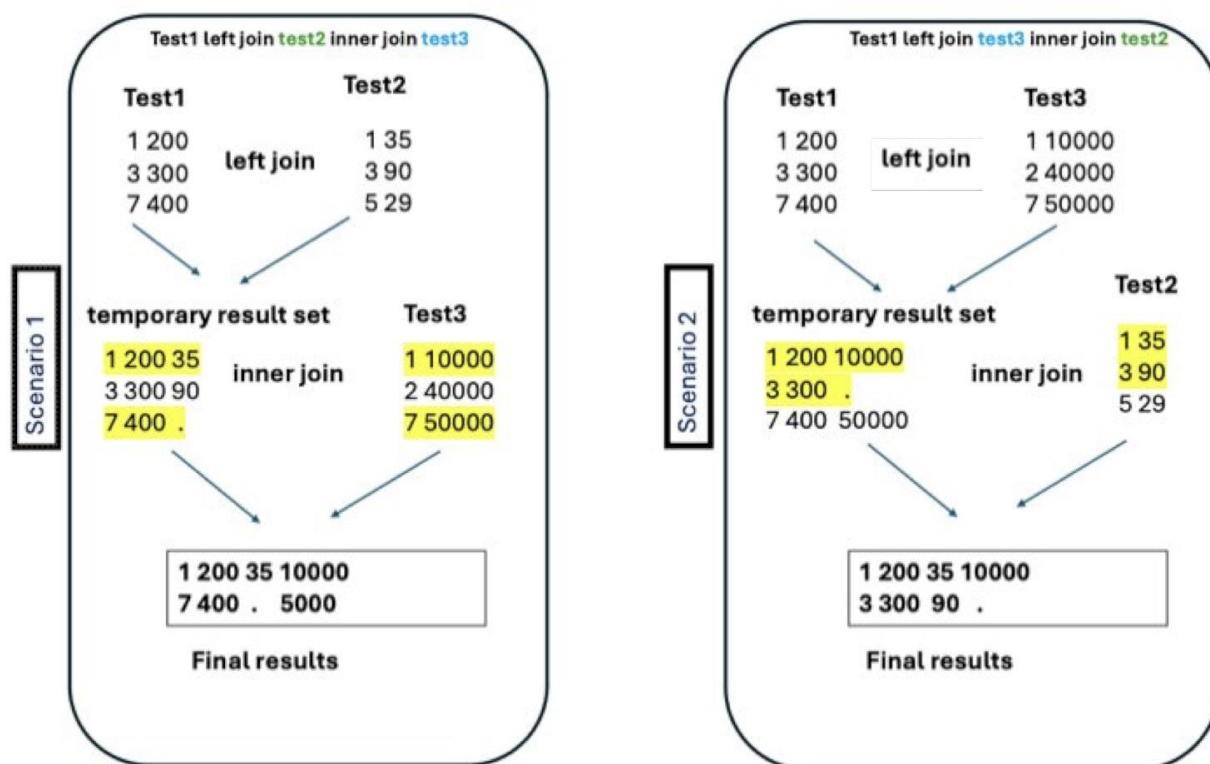


Figure 2: order of join execution with multi-table joins.

To conclude, it's important to bear in mind that the order of tables in multi-table joins can significantly impact the results. Always check the final output and make necessary adjustments to meet business requirements.

Unexpected outcomes in a LEFT JOIN when a WHERE clause is used instead of an AND condition.

To understand this scenario, it is important to first understand what happens in a typical LEFT JOIN. In LEFT JOIN, we retrieve all rows from the left (or first) dataset mentioned in the join, along with the matching rows from the right table. Let's illustrate this with the sample datasets shown below.

```
data one_3;
    input id name $;
    datalines;
1 Smith
2 John
3 Sam
;
run;

data two_3;
    input id age;
    datalines;
1 30
2 40
4 45
;
run;
```

In the code below, a LEFT JOIN is performed based on the ID column:

```
proc sql;
select a.*
      ,age
from one_3 a
left join two_3 b
on a.id=b.id;
quit;
```

From the above code, it is clear that we retrieve all variables from dataset one_3 and the age variable from dataset two_3. Whenever there is a match between the joining columns, the age from dataset two_3 is included; otherwise, it will be missing, as shown in the result below.

id	name	age
1	Smith	30
2	John	40
3	Sam	.

Now, let us see what happens when we add an additional join condition using an AND condition. In the code above for the LEFT JOIN, we are adding the condition age = 40, as highlighted in yellow, as shown below.

```
proc sql;
select a.*
      ,b.age
from one_3 a
left join two_3 b
on a.id=b.id
and age=40;
quit;
```

This typically extends what happens in a LEFT JOIN, where we retrieve everything from the left table and join values from the right table based on the ID and age variables. We get values from dataset two_3 only when the ID matches and the age is 40, and all other values for the age variable will be null, as shown below.

id	name	age
1	Smith	.
2	John	40
3	Sam	.

Now, if we modify the AND condition with a WHERE clause as shown below:

```
proc sql;  
select a.*  
  ,b.age  
from one_3 a  
left join two_3 b  
on a.id=b.id  
where age=40;  
quit;
```

Modifying the AND condition with a WHERE clause can yield entirely different results. In this case, we now have only one row instead of all the rows from the LEFT JOIN with the previously mentioned AND condition.

id	name	age
2	John	40

In a LEFT JOIN, we expect to see all rows from the left table. However, when a WHERE clause is applied, it filters the results based on the join condition. This happens in two steps:

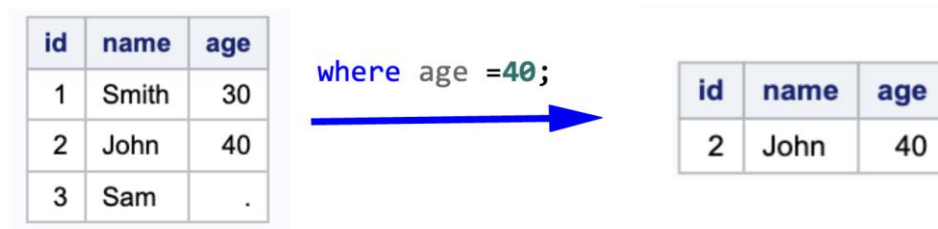
1. **Joining:** The tables are joined based on the specified conditions, creating a temporary result set.
2. **Filtering:** The WHERE clause is then applied to this temporary result set, filtering out rows that do not meet the condition specified in the WHERE clause.

This sequence illustrates how the WHERE clause operates after the join, influencing which rows are included in the result.

In first step join condition works and give the results as shown below.



In step 2, the results created from the above step are filtered by applying the WHERE clause. This filtering is based on the join condition specified in the ON clause and the additional conditions specified in the WHERE clause.



To conclude, we need to be careful when using the AND condition and the WHERE clause, and choose one over the other based on the specific business requirements.

Issue of joining datasets without considering duplicates in the joining column in both datasets

This scenario often occurs due to a lack of knowledge about the datasets and when the joining column has duplicates. Now, let us look into the datasets we want to join and understand the business requirement for the join.

```

data purchase;
    input product $
           purchase_amount
           purchase_date:date9.;
    format purchase_date date9.;
    datalines;
Candy 5 01JAN2024
Candy 5 02JAN2024
Candy 5 03JAN2024
;
run;

data sold;
    input product $
           sold_amount
           sold_date:date9.;
    format sold_date date9.;
    datalines;
Candy 7 04JAN2024
Candy 7 05JAN2024
Candy 7 06JAN2024
;
run;

```

From the above examples, we have a purchase dataset containing variables for product, purchase_amount, and purchase_date, and a sold dataset containing variables for product, sold_amount, and sold_date. The business requirement is to create a final dataset with the total purchase_amount and total sold_amount for each product. One strategy is to join on the product column from both datasets and sum the purchase_amount and sold_amount by product. This can be done as shown below.

```

proc sql;
select p.product,
       sum(purchase_amount) as total_purchase_amount,
       sum(sold_amount) as total_sold_amount
from purchase as p
inner join sold as s
on p.product =s.product
group by p.product;
quit;

```

The above code produces the results shown below. If you look closely, the total purchase_amount is much larger than expected. We had only 3 observations for 5 units, so our total_purchase_amount should be 15. Similarly, we had 3 observations for 7 units, so our total_sold_amount should be 21. The results indicate that the amounts are three times larger than expected, suggesting that something is wrong.

product	total_purchase_amount	total_sold_amount
Candy	45	63

The SQL query execution for the above code happens in two steps. The first step involves the join, and the next step performs aggregation using the SUM function. Now, let us look into the first step of execution where the join happens, along with the result, as shown below. The result shows a many-to-many join that results in a Cartesian product.

```
proc sql;
select p.product,
       purchase_amount,
       sold_amount
from purchase as p
inner join sold as s
on p.product =s.product;
quit;
```



product	purchase_amount	sold_amount
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7
Candy	5	7

In the next step, the group by product and aggregation occurs, giving the result discussed earlier.

To avoid this kind of situation, it is crucial to check for duplicates in the joining columns of the tables used for the join. For such scenarios, we need to aggregate each table first to ensure that each product has a unique value. Then, joins can be performed on these aggregated datasets. This approach is illustrated in the figure 3 shown below.

Aggregation first for purchase table by product as shown code below

```
proc sql;
create table total_purchase_byproduct as
select product,
       sum(purchase_amount) as total_purchase_amount
from purchase
group by product;
quit;
```

Gives unique value by Joining column AKA product

product	total_purchase_amount
Candy	15

Aggregation first for sold table by product as shown code below

```
proc sql;
create table total_sold_byproduct as
select product,
       sum(sold_amount) as total_sold_amount
from sold
group by product;
quit;
```

Gives unique value by Joining column AKA product

product	total_sold_amount
Candy	21

Joining table after aggregation

```
proc sql;
select p.product,
       total_purchase_amount,
       total_sold_amount
from total_purchase_byproduct as p
inner join total_sold_byproduct as s
on p.product =s.product;
quit;
```

Gives expected output

product	total_purchase_amount	total_sold_amount
Candy	15	21

Figure 3: Aggregation followed by Joins

Unexpected results when using joins based on the INDEX function or the LIKE operator

This scenario often occurs when joining two tables on character columns, where one column partially matches another column or vice versa. Let's understand this through an example with two datasets: one_5_full and two_5_partial as shown below.

```
data one_5_full;
input var1 $ var2;
datalines;
station 10
nation 70
cation 60
;
run;
```

```
data two_5_partial;
input col1 $2. col2;
datalines;
at 10
on 90
;
run;
```

We will join var1 from the one_5_full dataset with col1 from the two_5_partial dataset, where col1 is a substring within var1. This join can be done using the INDEX function in PROC SQL, resulting in a join condition different from regular

simple joins. Below is the code for same:

```
proc sql;
select      full.*
           ,partial.*
from        one_5_full as full
inner join  two_5_partial as partial
on          index(var1, col1) gt 0;
quit;
```

Queries like the one above can lead to many-to-many join. This is because the strings 'at' and 'on' in col1 of two_5_partial were found in all rows of var1. The results pasted below are from the PROC SQL query where joins were based on the INDEX function.

var1	var2	col1	col2
station	10	at	10
nation	70	at	10
cation	60	at	10
station	10	on	90
nation	70	on	90
cation	60	on	90

When using INDEX function for joining datasets, it's straightforward to understand with simple samples like the one I've taken. However, it can become tricky with larger datasets. Joining based on INDEX function should be minimized because it often results in unexpected outcomes, especially due to many-to-many join.

Conclusions

Understanding how PROC SQL joins are executed is crucial for avoiding unusual results. Unexpected outcomes can arise from missing values, many-to-many join,

and the position of table in multiple joins. It's also important to review the outputs after performing the joins to ensure accuracy.

Acknowledgements

I would like to thank, Pooja Joshi for helping me with proof reading and giving valuable suggestions.

Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Kiran Venna

Email: kvenna.sas@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.