Learning SAS PROC SQL: How to Use the EXCEPT Operator for Data Comparison

Authored by

Mohammed looti

April 29, 2026

Introducing the EXCEPT Operator for Data Differentiation in PROC SQL

Data integrity and comparison are cornerstones of effective data management and advanced analytics. When working within the **SAS** environment, particularly utilizing the powerful set of tools provided by **PROC SQL**, the **EXCEPT operator** emerges as an essential utility. This operator is fundamentally designed to execute a set difference operation, returning only those unique rows that are present in the result set of the first query but completely absent from the result set of the second. This capability is paramount for numerous business and analytical tasks, ranging from rigorous data validation and auditing records to identifying subtle discrepancies or tracking version changes across large volumes of information stored in different **datasets**.

The ability to accurately and efficiently pinpoint records unique to one source, while excluding all common elements, is often vital for maintaining the high quality and reliability of enterprise data. By fully embracing the structure and flexibility of the **SQL**-like syntax integrated into **PROC SQL**, users gain the facility to perform sophisticated, row-level comparisons with minimal coding complexity. This comprehensive guide aims to explore the practical implementation of the **EXCEPT operator**, providing clear, executable examples and offering valuable insights into its most effective deployment across diverse analytical landscapes within the **SAS** system.

Core Principles of Set Operators in SQL and SAS

To truly appreciate the utility of the **EXCEPT operator**, it is critical to first grasp the overarching concept of **set operators** within the **SQL** framework. These operators are fundamentally designed to logically combine the outcomes of two or more independent **SELECT statements** into a singular, unified result set. Beyond **EXCEPT**, the most commonly encountered **set operators** include `UNION`, which aggregates all distinct rows from both queries, and `INTERSECT`, which restricts the output to only those distinct rows that are shared between both queries. The **EXCEPT operator**, conversely, serves the specific purpose of isolating and returning the difference between the two sets.

The operational foundation of all **set operators** is rooted in mathematical set theory. When utilizing the **EXCEPT operator**, the underlying logic is a direct request for all elements belonging to the first set (or query result) that are definitively not members of the second set. For this difference operation to be successfully executed and yield meaningful results, both involved **SQL** queries must strictly adhere to specific structural mandates. Chief among these requirements is the necessity for both queries to return an identical number of columns, and, furthermore, that the corresponding columns in each query must possess compatible data types to allow for a valid comparison.

Ensuring column compatibility is not merely a technical requirement but a logical one; it guarantees

that a reliable row-by-row comparison can be accurately performed. If the column counts or data types diverge, **PROC SQL** will not be able to establish equivalence between the rows, leading to errors or inconsistent results. By adhering to these foundational rules, developers ensure that **SAS** can meticulously identify and isolate the records that are genuinely unique to the first specified **dataset**, providing a precise measure of the set difference.

Preparing Input Data for Set Difference Analysis in SAS

To illustrate the exact mechanics of the **EXCEPT operator**, we must first establish a controlled environment by generating two distinct sample **datasets** within **SAS**. These examples will contain simple, comparable information--specifically, basketball player records including their team affiliation and points scored--which will allow us to immediately see the results of the set operations. This deliberate setup, involving both overlapping and unique records, is crucial for demonstrating the precise filtering power of the operator.

We begin by constructing the initial **dataset**, designated as `data1`. This construction is achieved using a standard **DATA step**, where we formally define the variables `team` (a character variable) and `points` (a numeric variable) using the **input statement**. The data itself is then directly supplied within the **datalines** block. Once the data creation is complete, we employ the **proc print** procedure to display the contents of `data1`, verifying that the data has been loaded correctly and is ready for subsequent comparison operations.

```
/*create first dataset*/
data data1;
input team $ points;
datalines;
A 12
A 14
A 15
A 18
A 20
A 22
;
run;

/*view first dataset*/
proc print data=data1;
```

Obs	team	points
1	A	12
2	A	14
3	A	15
4	A	18
5	A	20
6	A	22

Following the creation of our first set, we proceed to construct the second **dataset**, named `data2`, utilizing the identical methodology to ensure structural compatibility. This second set is populated with player data that intentionally includes two records shared with `data1` (`A 12` and `A 14`) and several records unique to `data2`. This deliberate overlap is essential for showcasing the filtering accuracy of the set difference operation we are about to perform. After `data2` is successfully created, we use **proc print** one last time to visually confirm its contents, thereby setting the stage for the powerful application of the **EXCEPT operator**.

```
/*create second dataset*/
data data2;
input team $ points;
datalines;
A 12
A 14
B 23
B 25
B 29
B 30
;
run;

/*view second dataset*/
proc print data=data2;
```

Obs	team	points
1	A	12
2	A	14
3	B	23
4	B	25
5	B	29
6	B	30

Executing the EXCEPT Operation: Identifying Rows Unique to the First Dataset (data1 EXCEPT data2)

With our two structured **datasets**, `data1` and `data2`, now prepared and verified, we are ready to implement the core functionality of the **EXCEPT operator**. Our first and most direct application is aimed at isolating all records exclusively found in `data1`, meaning any row that does not have an exact, corresponding match within `data2` will be returned. This objective is fulfilled by constructing a **PROC SQL** query block that seamlessly connects two distinct **SELECT statements** using the `EXCEPT` keyword.

The required syntax for this operation is remarkably concise yet powerful. We initiate the process with the first **SELECT statement**, specifying that we wish to retrieve all columns (`SELECT *`) **from** `data1`. This is immediately followed by the **EXCEPT operator**, which then precedes the second **SELECT statement** targeting `data2`. Upon execution, **SAS** initiates a meticulous comparison process: every row in `data1` is checked against every row in `data2`. If a row in the primary set (`data1`) finds an exact match across all column values in the exclusion set (`data2`), that row is efficiently filtered out. Only the rows that stand alone in `data1` are permitted into the final result set.

```
/*only return rows from first dataset that are not in second dataset*/
proc sql;
title 'data1 EXCEPT data2';
select * from data1
except
select * from data2;
quit;
```

data1 EXCEPT data2

team	points
A	15
A	18
A	20
A	22

When scrutinizing the resulting output, the effectiveness of the operation is immediately clear: only the rows `A 15`, `A 18`, `A 20`, and `A 22` are displayed. This outcome confirms that the initial shared records--`A 12` and `A 14`--which existed in both `data1` and `data2`, were correctly identified and removed by the **EXCEPT operator**. This successful demonstration highlights the operator's exceptional precision in calculating asymmetrical differences between two or more complex data sources.

Understanding Directionality: Finding New Entries (data2 EXCEPT data1)

A critical feature that distinguishes the **EXCEPT operator** from symmetrical operations like `UNION` is its inherently directional nature. The sequence in which the **SELECT statements** are positioned dramatically dictates the final result set. In our previous example, `data1 EXCEPT data2` isolated rows unique to the first set. Conversely, if we deliberately reverse the order of the inputs to execute `data2 EXCEPT data1`, the query shifts its focus entirely, requesting all rows that are present exclusively in `data2` while excluding any matches found in `data1`.

This directional property transforms **EXCEPT** into an exceptionally flexible tool for specific data reconciliation and versioning tasks. Consider scenarios involving temporal data: if `data1` represents the records from last month and `data2` represents the records from the current month, the operation `data1 EXCEPT data2` would efficiently identify all records that were removed or deleted during the intervening period. Conversely, performing `data2 EXCEPT data1` would quickly highlight all newly added or inserted records. This ability to define the comparison direction is invaluable for change tracking.

To observe this directional shift in practice, we modify our existing **PROC SQL** query, placing `data2` as the primary set and `data1` as the exclusion set. This small change in syntax fundamentally alters the analytical question being posed to the system, now focusing on records that only appeared in our second data source.

/*only return rows from second dataset that are not in first dataset*/

```
proc sql;
title 'data2 EXCEPT data1';
select * from data2
except
select * from data1;
quit;
```

data2 EXCEPT data1

team	points
B	23
B	25
B	29
B	30

As expected, the output of `data2 EXCEPT data1` clearly presents rows `B 23`, `B 25`, `B 29`, and `B 30`. These are precisely the entries that exist solely within `data2` and have no identical match in `data1`. Crucially, the common rows (`A 12` and `A 14`) are once again correctly excluded. This outcome provides a compelling illustration of how the strict ordering of the **SELECT statements** when using **EXCEPT** determines which unique records are returned, making it imperative to define the comparison direction based explicitly on the analytical objective.

Advanced Considerations and Best Practices for EXCEPT Implementation

While implementing the **EXCEPT operator** within **PROC SQL** is generally straightforward, adhering to several best practices and understanding specific technical nuances can significantly enhance efficiency and prevent potential pitfalls. The absolute foundational rule is ensuring robust compatibility between the two queries: the number of columns must match, and the data types of corresponding columns must be consistent. Although using `SELECT *` is convenient, explicitly listing columns (e.g., `SELECT Team, Points`) is highly recommended. Explicit listing offers greater clarity, maintains control over the column order, and mitigates risks if the underlying schema of one of the **datasets** is unexpectedly altered.

Another crucial consideration in **SAS** **SQL** operations is the treatment of missing values. By default, when **PROC SQL** compares rows, two missing values are considered equal. This means that if a row in `data1` contains missing values that exactly match the missing values pattern in a row in `data2`, that row will be treated as a duplicate and consequently filtered out by the **EXCEPT**

operator. Analysts must be acutely aware of their data's missingness patterns. For optimal reliability, it is often advisable to pre-process data to handle, standardize, or explicitly exclude records with missing values before performing set operations, thereby ensuring the comparison results truly reflect structural differences rather than null equivalence.

Finally, performance optimization becomes a relevant factor when dealing with exceptionally large **datasets**. While **PROC SQL** is highly optimized for set operations, alternative approaches exist. For instance, achieving the same set difference result can be accomplished using a traditional `LEFT JOIN` combined with a restrictive `WHERE` clause (specifically, `WHERE second_table.key IS NULL`). For tables with established indexes on the comparison keys, this join method can occasionally outperform the set operator syntax. Nevertheless, for the majority of routine data comparison tasks, the **EXCEPT operator** provides the most readable, concise, and typically efficient method available within the **SAS** analytical framework.

Conclusion: Mastering Data Differentiation with EXCEPT

The **EXCEPT operator**, when utilized within **PROC SQL**, stands as an indispensable and highly effective tool for any analyst working in the **SAS** environment who requires precise data cleansing, validation, and complex reconciliation. By enabling the accurate identification of unique rows that exist in one **dataset** but not another, this operator empowers users to rapidly detect anomalies, track incremental changes, and systematically ensure the fidelity and integrity of their critical information assets. Its elegant, straightforward syntax, combined with the underlying robustness of the **PROC SQL** engine, establishes it as a superior method for many complex data comparison tasks.

Mastering the directional nature of **EXCEPT**--understanding the difference between `A EXCEPT B` and `B EXCEPT A`--and strictly adhering to best practices concerning column compatibility and consistent data handling (especially regarding missing values) are key steps toward ensuring reliable and repeatable analytical results. Whether your task involves auditing transactional logs, comparing versions of master customer lists, or reconciling large financial records, the **EXCEPT operator** provides a clear, powerful, and efficient mechanism to reveal the subtle yet critical distinctions within your data.

Further Learning and Resources

To continue expanding your proficiency in **SAS** programming and advanced **SQL** techniques, it is highly beneficial to study the complementary **set operators**, specifically `UNION` and `INTERSECT`, and how they function within the **PROC SQL** procedure. The following resources offer additional tutorials for performing common data manipulation and analysis tasks in **SAS**:

Tutorial on using the **UNION operator** to combine distinct rows.

Guide to implementing the **INTERSECT operator** to find common rows.

Deep dive into optimizing **PROC SQL** queries for large-scale data processing.