Perform an Anti-Join in PySpark

Authored by

stats writer

November 16, 2025

Introduction to Relational Joins and PySpark

Data manipulation in modern big data environments, particularly those utilizing PySpark, frequently requires combining or comparing information across multiple datasets. The fundamental mechanism for achieving this is the join operation, a concept borrowed heavily from relational database management systems and SQL. While standard joins like inner and outer joins focus on retrieving matching or inclusive records, specialized join types are essential for exclusion scenarios.

The data structure central to these operations in Spark is the DataFrame. A PySpark **DataFrame** is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database. When dealing with massive datasets, selecting the correct join type is paramount to ensuring computational efficiency and obtaining accurate results.

Understanding the Anti-Join Operation

The **anti-join**, officially implemented as a left anti join in PySpark, serves a very specific purpose in data filtering: exclusion. Rooted in the principles of set theory and Relational Algebra, the anti-join operation provides a highly optimized way to identify and return all rows from the primary, or left, **DataFrame** that do not have any corresponding match in the secondary, or right, **DataFrame** based on the specified join key(s).

In essence, if you join DataFrame A to DataFrame B using a `left_anti` join, Spark processes A and B and returns only the rows from A that are unique with respect to B. The resulting schema is always identical to the left **DataFrame**, as no columns are retrieved from the right side. This makes the anti-join superior for scenarios where you are simply filtering a list based on non-existence in another list, rather than retrieving related data.

Key benefits of using an **anti-join** include enhanced performance over alternative exclusion methods (like performing a left outer join and then filtering for nulls) and superior code clarity. It is the definitive method in PySpark for achieving a set difference operation.

Syntax for Performing an Anti-Join in PySpark

To execute this powerful exclusion logic, PySpark utilizes the standard `.join()` method available on every **DataFrame** object. The critical distinction lies in defining the `how` parameter as `'left_anti'`. This parameter explicitly instructs the Spark engine to apply the anti-join filtering logic.

You can use the following standard syntax to perform an **anti-join** between two PySpark DataFrames, assuming `df1` is the left DataFrame (the source of the output rows) and `df2` is the

right DataFrame (the source of the exclusion criteria):

df_anti_join = df1.join(df2, on=, how='left_anti')

This example demonstrates a fundamental filtering operation. The output DataFrame, `df_anti_join`, will only contain the rows originating from `df1` where the value in the specified join column, `team`, does not successfully find a match within the `team` column of `df2`. This approach is highly performant because Spark can often optimize the anti-join execution plan by utilizing semi-join transformations under the hood.

Example: Setting up the PySpark Environment and DataFrames

To illustrate the functionality of the **anti-join**, we will set up a working environment and define two sample DataFrame objects. This setup ensures we have a context where teams exist in one list but are missing from the other, allowing us to clearly see the exclusion mechanism in action.

We begin by initiating the SparkSession, the necessary gateway for using the PySpark API, and then defining our first dataset, `df1`, which acts as our universe of records.

Defining Source DataFrame 1: The Primary Dataset (df1)

Our first **DataFrame**, `df1`, represents a collection of team names and their corresponding point totals. This DataFrame contains five unique teams (A, B, C, D, E) and serves as the dataset from which we wish to remove any matching entries found in the exclusion list (`df2`).

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = ,
,
,
,
]

#define column names
columns1 =

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

```
#view dataframe
df1.show()

+----+------+
|team|points|
+----+------+
| A| 18|
| B| 22|
| C| 19|
| D| 14|
| E| 30|
+----+------+
```

Defining Source DataFrame 2: The Exclusion List (df2)

Next, we define `df2`, which acts as the reference list containing elements that should be excluded from `df1`. Notice that teams A, B, and C exist in both DataFrames, while teams D and E exist only in `df1`. Teams F and G exist only in `df2`, but their presence is irrelevant to the output since the anti-join only returns rows from the left side (`df1`).

```
#define data
data2 = ,
,
,
]

#define column names
columns2 =

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+----+------+
|team|points|
+----+------+
| A| 18|
| B| 22|
```

C	19
F	22
G	29
+----+------+

Our goal is to isolate the rows in `df1` that correspond to teams D and E, as those are the only records that do not find a matching key in `df2`.

Executing the PySpark Anti-Join and Analyzing Results

We are now ready to apply the **anti-join**. By specifying `how='left_anti'`, we instruct PySpark to perform the efficient lookup and exclusion based on the shared `team` column.

The following code snippet performs the join and immediately displays the resulting **DataFrame**, confirming which records were successfully filtered out and which remained.

#perform anti-join
df_anti_join = df1.join(df2, on=, how='left_anti')

#view resulting DataFrame
df_anti_join.show()

+----+------+
|team|points|
+----+------+
| D| 14|
| E| 30|
+----+------+

The output confirms that the operation functioned exactly as expected. The resulting **DataFrame**, `df_anti_join`, contains only the rows for teams D and E. Teams A, B, and C were successfully excluded because their `team` values found a corresponding match in `df2`. This demonstrates the power and simplicity of using `left_anti` for efficient data exclusion.

Summary of Anti-Join Utility

The **anti-join** is a fundamental and highly optimized tool for data engineers working with big data distributed processing. It offers a concise, readable, and highly efficient mechanism for solving exclusion problems that would otherwise require complex subqueries or multi-step filtering logic.

By returning only the unique rows from the left DataFrame that fail to find a match in the right DataFrame, the anti-join ensures that filtering operations are performed with minimal overhead, making it indispensable for maintaining robust and scalable data pipelines in Apache Spark environments.