How to Find Rows in One PySpark DataFrame That Are Not in Another DataFrame

Authored by

stats writer

February 5, 2026

PySpark: Get Rows Which Are Not in Another DataFrame

The Challenge of Data Differencing in Big Data Environments

In the realm of Big Data processing, comparing and contrasting massive datasets is a fundamental requirement for tasks ranging from data validation to change data capture (CDC). When working with distributed systems like Apache Spark, simply iterating and comparing records is computationally inefficient, often leading to severe bottlenecks. A common and critical challenge faced by data engineers is efficiently determining which records exist in a primary dataset but are entirely missing from a secondary, comparison dataset. This operation, which is mathematically analogous to a set difference, demands specialized functions optimized for distributed PySpark DataFrame manipulation to ensure accuracy and speed across cluster nodes.

Traditional relational database management systems often rely on conditional `LEFT JOIN`s combined with filtering on `NULL` values to achieve this difference identification. While Spark SQL permits this strategy, leveraging native DataFrame transformation methods, such as those designed for set operations, typically provides cleaner, more concise syntax and superior performance due to deep optimization within Spark's Catalyst optimizer. It is essential to select the function specifically designed for this purpose to write truly robust and scalable Spark applications. This method must correctly handle data partitioning and accurately preserve the counts of duplicate records when determining the difference set.

This article focuses on the specific and highly recommended tool provided by the PySpark API to tackle this problem: the `exceptAll` transformation. This powerful method allows practitioners to easily and efficiently obtain the exact subset of rows from one PySpark DataFrame that are wholly absent in a second PySpark DataFrame. The combination of precision and high performance offered by `exceptAll` makes it an indispensable technique for advanced data analysis, migration verification, and integration workflows where exact row matching is mandatory.

Introducing the `exceptAll` Method for Set Difference

To effectively isolate the rows from a source DataFrame that are not contained within a comparison DataFrame, data engineers should employ the `exceptAll` function. This function executes a distributed, multiset difference operation. The critical distinction of `exceptAll`, compared to the older PySpark `except` method, is its capability to respect the multiplicity of rows. This means it accurately accounts for duplicate records when calculating the final difference. For example, if a record appears five times in the primary DataFrame (A) and twice in the comparison DataFrame (B), `exceptAll` will correctly return that record three times (5 - 2 = 3), reflecting the

true delta in count.

The operational mechanism of the `exceptAll function` mandates a comprehensive comparison of all column values across every row in both input DataFrames. For a row from the primary DataFrame to be excluded from the result set, it must match an identical row, including considering the number of times it appears, in the secondary DataFrame. This meticulous filtering ensures that the resulting DataFrame represents only those rows that are truly unique or have an excess count in the source relative to the comparison set. Such strict comparison capabilities are vital when managing time-series data, audit trails, or complex transactional records.

This transformation offers superior efficiency for isolating disparities compared to writing custom, complex join and filtering expressions. Furthermore, the `exceptAll function` is part of a suite of native Set operations in PySpark, which also includes `intersectAll` and `unionAll`. This integration provides a complete and optimized toolkit for distributed data manipulation, allowing developers to leverage well-established mathematical principles directly in their data pipelines. Mastering these set-based transformations leads directly to cleaner, more maintainable, and highly optimized Spark codebases.

Syntax and Practical Application of `exceptAll`

The syntax required to execute the set difference operation using `exceptAll` is remarkably simple, fitting seamlessly into the fluid, object-oriented style of the PySpark DataFrame API. This method promotes readability and expresses complex logic in a single, self-explanatory line of code, which significantly benefits collaboration and debugging across large development teams.

You can use the following syntax to get the rows in one PySpark DataFrame which are not in another DataFrame:

df1.exceptAll(df2).show()

This concise expression demonstrates the correct usage: the `exceptAll` method is invoked on the primary DataFrame (`df1`), and the comparison DataFrame (`df2`) is passed as the sole argument. The output is a new DataFrame containing the set difference A minus B. Applying `.show()` immediately after the transformation triggers the execution and displays the resulting rows. It is imperative to remember that this set difference is asymmetrical; `df1.exceptAll(df2)` focuses on records unique to `df1`, while the reverse operation, `df2.exceptAll(df1)`, would isolate records unique to `df2`.

The central advantage here is the functional clarity: this method returns all of the rows from the DataFrame named **df1** that are not in the DataFrame named **df2**, including preserving the counts of any duplicates that are only present in `df1`. This functional abstraction eliminates the manual

complexity associated with constructing equivalent logic using distributed hash or merge joins, thereby reducing the probability of errors and accelerating development cycles. This simple syntax provides immediate access to high-performance difference calculations across massive distributed datasets.

Step-by-Step Example: Defining Source DataFrames

To illustrate the functionality of `exceptAll`, we will define two distinct sample DataFrames. This practical demonstration will clearly show how the function handles common and unique rows. We begin by establishing the necessary Spark context, which is achieved through the `SparkSession` object, the fundamental starting point for all PySpark operations.

Suppose we have the following DataFrame named **df1**, representing an initial dataset of records:

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

+----+------+

DataFrame `df1` contains five total rows. We can identify that the rows ('A', 18), ('B', 22), and ('C', 19) are potentially shared records, while ('D', 14) and ('E', 30) are unique entries we expect to isolate. Defining these expectations beforehand is crucial for validating the output of the `exceptAll` operation.

Next, we introduce the secondary comparison DataFrame, **df2**. Before proceeding with any Set operations in PySpark, it is a strict requirement that the schemas of both DataFrames are fully compatible--they must share identical column names, ordering, and data types. This structural integrity is non-negotiable for guaranteeing the accuracy of the distributed row comparison.

And suppose we have another DataFrame named **df2**:

#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
+----+------+

Implementing the Set Difference Operation

With both DataFrames initialized and their schemas aligned, we can proceed to execute the core set difference logic using `exceptAll`. This single operation calculates the exact difference between the two distributed datasets. The efficiency of this function stems from the fact that Spark utilizes highly optimized shuffle mechanisms to group and compare records across the cluster, rather than relying on less efficient methods like broad joining.

We can use the following syntax to return all rows that exist in **df1** that do not exist in **df2**:

#display all rows in df1 that do not exist in df2
df1.exceptAll(df2).show()

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

The resulting DataFrame confirms that only the rows ('D', 14) and ('E', 30) are returned. We can see that there are exactly two rows from the first PySpark DataFrame that do not exist in the second DataFrame. This validates the intended behavior of the function, which is to precisely isolate records unique to the source DataFrame (`df1`) relative to the comparison DataFrame (`df2`). This result set is often referred to as the delta or change set.

Understanding the performance implications of `exceptAll` is also vital. Because it requires a comparison across all columns and respects duplicates, it often involves a wide transformation (shuffling data across partitions). Proper optimization, such as choosing the right partitioning scheme for the DataFrames involved, can dramatically enhance the speed of the difference calculation, making it scalable for truly colossal datasets.

Key Differences: `except` vs. `exceptAll`

Prior to Apache Spark version 2.3, the only function available for set difference was `except()`. While functional, `except()` operates under the rules of standard set theory, behaving like a SQL `EXCEPT DISTINCT` clause. This means that `except()` implicitly removes duplicate rows from the final result set, focusing only on the existence of distinct records. This behavior is problematic for tasks like financial auditing or CDC where the exact count of duplicates (multiplicity) is integral to the data meaning.

The primary divergence is the handling of duplicates. If `df1` contains three instances of Record R, and `df2` contains one instance of Record R: `df1.except(df2)` would return zero rows because the distinct record R exists in `df2`. In contrast, `df1.exceptAll(df2)` would correctly return two instances of Record R, accurately reflecting the mathematical difference in count (3 minus 1).

Data engineers must consciously decide which function to use based on the data integrity requirements. If the goal is merely to identify which distinct record types are unique to a dataset, `except()` may suffice. However, if the requirement is for rigorous reconciliation, data synchronization, or auditing where every instance of a row must be tracked and accounted for, the use of `exceptAll` is absolutely essential. It guarantees fidelity to the original source dataset counts, providing a far more accurate representation of the operational delta between the two distributed datasets.

Advanced Use Cases and Best Practices

The utility of `exceptAll` extends into sophisticated data pipeline architectures, particularly those involving iterative processing and high-stakes data quality checks. Beyond basic reconciliation, it is frequently used in scenarios like filtering known good records from a potentially flawed stream to isolate errors, or streamlining ETL processes by ensuring only new or unique records are passed downstream for costly processing steps.

Best practices dictate careful adherence to several guidelines when implementing `exceptAll`:

Schema Integrity: This remains the most critical rule. Always verify that `df1` and `df2` share identical column structure, order, and precise data types. Schema mismatches are the most common source of failure or logically incorrect results in set operations.

Performance Considerations: Because `exceptAll` is a wide transformation, it causes data shuffling. For maximum efficiency, consider pre-partitioning both DataFrames based on a common key if the underlying data allows, which can minimize the necessary shuffle during the comparison phase.

Integration with other Functions: The function works harmoniously with other operations. For instance, finding records that have been deleted from an updated snapshot (B) relative to an old snapshot (A) simply requires calculating `A.exceptAll(B)`. Meanwhile, new insertions are found via `B.exceptAll(A)`.

In summary, whenever the task requires accurately isolating the unique records and their counts from a primary dataset relative to a secondary dataset, the `exceptAll` method is the most reliable and performant choice within PySpark. Its optimization and strict adherence to multiset principles ensure accurate results necessary for enterprise data management.

Note: You can find the complete documentation for the PySpark **exceptAll** function here.

The following tutorials explain how to perform other common tasks in PySpark: