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

Authored by

stats writer

January 19, 2026

RECOMMENDED CITATION

stats writer (2026). *How to Find Rows in a PySpark DataFrame That Are Not in Another DataFrame*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=126637

ARABPSYCHOLOGY.COM

Understanding Set Difference in Big Data Contexts

In modern data processing, particularly within the realm of Apache Spark, data engineers frequently encounter the need to perform Set Operations on large datasets. One of the most fundamental requirements is determining the set difference: identifying which records exist in one dataset but are absent in another. This operation is crucial for tasks like detecting newly added transactions, identifying deleted users, or performing data validation checks between system snapshots.

When working with PySpark, data is typically managed within a distributed structure known as a **DataFrame**. Unlike simple SQL tables where primary keys often simplify comparisons, DataFrames can contain millions or billions of rows, and may often include duplicate entries, making the accurate determination of differences non-trivial. Achieving precise set difference requires a function that meticulously compares every column and respects the multiplicity of rows, ensuring that if a row appears three times in the first DataFrame and only once in the second, the two remaining occurrences are correctly identified as the difference.

The challenge in a distributed computing environment like Spark lies in efficiently executing these comparisons across multiple nodes without moving excessive amounts of data. This is where PySpark provides specialized, optimized functions designed to handle these massive operations effectively, minimizing the need for complex, manual join logic that could easily lead to performance bottlenecks or incorrect results if not handled precisely.

The Primary PySpark Solution: Utilizing exceptAll()

To effectively retrieve all rows present in a source DataFrame (`df1`) that do not exist in a target DataFrame (`df2`), PySpark provides the powerful and straightforward `exceptAll()` transformation. This method performs a true set difference operation, returning a new DataFrame containing only the distinct rows from the first DataFrame that are entirely absent from the second. Critically, `exceptAll()` respects the existence of duplicate records, which is essential for accurate data auditing and complex ETL (Extract, Transform, Load) pipelines.

The syntax is remarkably concise, reflecting the declarative nature of Spark programming. Assuming you have two DataFrames, `df1` and `df2`, the operation to find the unique records in `df1` is simply chaining the function call:

df1.exceptAll(df2).show()

This execution immediately triggers the necessary distributed computation within the Apache Spark cluster. The resulting output will be a DataFrame subset of `df1`, containing only those rows

ARABPSYCHOLOGY.COM

that failed to find an exact match in `df2`. It is vital to remember that this operation is strictly asymmetrical; `df1.exceptAll(df2)` does not yield the same result as `df2.exceptAll(df1)`, as the latter would return rows unique to the second DataFrame.

Distinguishing exceptAll() from except_()

When exploring PySpark documentation, users might encounter both `exceptAll()` and the older `except_()` function. Understanding the subtle but critical difference between these two methods is paramount for ensuring data integrity, especially in environments where data duplication is common or expected. The key distinction lies in how each function handles duplicate rows present within the DataFrames.

The legacy function, `except_()`, behaves similarly to a standard SQL `EXCEPT DISTINCT` clause. If `df1` contains a row (A, 10) three times, and `df2` contains that row (A, 10) once, `except_()` treats the operation as set difference based on distinct values. Thus, the row (A, 10) would be entirely eliminated from the result set because it appeared at least once in `df2`. This behavior is ideal when the goal is to find differences based on unique keys or records, disregarding counts.

In contrast, the `exceptAll()` function, introduced in later versions of Spark (2.3+), respects the multiset semantics--meaning it adheres to the count of duplicates. Using the same example (A, 10) appearing three times in `df1` and once in `df2`, exceptAll will correctly return the row (A, 10) twice in the resulting DataFrame, as two instances of that row were not accounted for in `df2`. Because of this robust handling of duplicates, `exceptAll()` is the recommended approach for achieving an accurate, multiset difference between two DataFrames.

Practical Implementation: Setting Up the DataFrames

To illustrate the functionality of exceptAll, we must first establish a working environment and define two sample DataFrames. We begin by initializing a **SparkSession**, the entry point for using Spark functionality, and then define our data arrays and schema. The column names must be identical between `df1` and `df2` for the `exceptAll()` operation to execute successfully, as the comparison is performed column-by-column across the entire row structure.

Consider the following setup, where `df1` represents an initial snapshot of team scores and `df2` represents a partial or updated list. Our goal is to isolate which scores only appear in the initial snapshot, `df1`. The data structure includes two fields: `team` (string identifier) and `points` (integer value).

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

ARABPSYCHOLOGY.COM

#define data for df1
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
+----+------+

Next, we define our second DataFrame, `df2`. Notice that `df2` shares the first three rows (A, B, C) with `df1` but introduces two entirely new rows (F and G). Crucially, rows D and E from `df1` are absent in `df2`, setting up the scenario where `exceptAll()` will correctly identify them as the difference.

#define data for df2
data2 = ,
,
,
,
]

#define column names
columns2 =

ARABPSYCHOLOGY.COM

```
#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|
+----+------+
```

Executing the Row Difference Operation

With both DataFrames initialized and containing overlapping and unique records, we can now execute the core operation to determine the set difference. By calling `df1.exceptAll(df2)`, we instruct PySpark to scan every row in `df1` and verify its existence in `df2` based on the exact match of all column values.

The efficiency of this operation is managed internally by Spark, often involving a shuffle operation to bring corresponding partitions of data together for comparison. This process ensures that even across a massively parallel architecture, the comparison is accurate and complete, making it highly reliable for large-scale reconciliation tasks.

Executing the following command yields the expected result:

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

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

ARABPSYCHOLOGY.COM

The output clearly shows the two rows, (D, 14) and (E, 30). These were the only records that were exclusively present in `df1` and had no exact corresponding match in `df2`. This result demonstrates the functionality of exceptAll in isolating truly unique records based on multiset comparison, fulfilling the requirement of finding rows from the first set that are not contained within the second set.

Alternative Methods for Finding Row Differences (Anti-Join)

While `exceptAll()` is the most direct and semantically accurate way to perform a multiset difference across the entire row, data engineers often use an alternative technique involving **Anti-Joins**, particularly when the comparison needs to be based only on a subset of key columns, or when performance tuning requires avoiding the full row comparison overhead.

An Anti-Join (implemented in PySpark as a `left_anti` join) returns only the rows from the left DataFrame for which the join key does not find a match in the right DataFrame. This can emulate a set difference, but the key difference is that the Anti-Join only requires matching on specified key columns, whereas exceptAll requires all columns to match. If we wanted to find rows in `df1` whose `team` identifier is not present in `df2`, regardless of the `points` value, an Anti-Join would be the appropriate choice.

For instance, if we used an Anti-Join based only on the `team` column, the code would look like: `df1.join(df2, on='team', how='left_anti').show()`. In our example, this would yield the same result (rows D and E) because their team identifiers are unique. However, if row 'A' in `df1` had 18 points and row 'A' in `df2` had 20 points, `exceptAll()` would keep row 'A' from `df1` (since the full row differs), but the Anti-Join would drop it (since the join key 'A' matched). Therefore, the choice between `exceptAll()` and Anti-Join hinges on whether you require a full row comparison (set difference) or a key-based exclusion.

Performance Considerations and Best Practices

When working with enormous DataFrames, performance is always a critical concern. Both `exceptAll()` and its alternatives are wide transformations in PySpark, meaning they require a shuffle operation. A shuffle involves serializing, transferring, and deserializing data across the network between executors, which is the most expensive operation in Spark.

To optimize the use of `exceptAll()`, data engineers should follow several best practices. Firstly, ensuring that the DataFrames used in the operation have matching schemas and data types is paramount, as mismatches can lead to runtime errors or incorrect comparisons. Secondly, if the DataFrames are highly skewed (i.e., some partitions contain vastly more data than others), the shuffle operation can become bottlenecked. Techniques like salting or repartitioning the

ARABPSYCHOLOGY.COM

DataFrames before the operation can help distribute the load more evenly across the cluster.

Finally, consider caching the DataFrames if they are to be used multiple times in complex workflows. Caching allows Spark to persist the data in memory (or on disk), preventing the need to recompute the DataFrames from their source every time the `exceptAll()` operation is invoked. While `exceptAll()` is highly optimized for set difference, understanding the underlying distributed nature of Apache Spark ensures efficient and scalable execution across massive datasets.

For comprehensive details and additional parameters, users should always consult the official PySpark API documentation for the exceptAll function.

Related PySpark Tutorials

Mastering Set Operations is just one step in data engineering with Apache Spark. The following tutorials explain how to perform other common tasks in PySpark:

How to perform efficient joins.

Methods for handling null values in DataFrames.

Techniques for aggregating and grouping data using window functions.

ARABPSYCHOLOGY.COM