How to use `where()` and `filter()` in a DataFrame with Examples

Filtering Rows in a Spark DataFrame: Techniques and Tips

Ahmed Uz Zaman Follow 4 min read · Jan 31, 2023

Photo by Bruno Wolff on Unsplash

Description

In Apache Spark, the `where()` function can be used to filter rows in a DataFrame based on a given condition. The condition is specified as a string that is evaluated for each row in the DataFrame. Rows for which the condition evaluates to `True` are retained, while those for which it evaluates to `False` are removed.

The basic syntax for the `where()` function is:

```
DataFrame.where(condition)
```

1. Filtering rows with one condition

Using **`where`** *condition:*

For example, the following code filters a DataFrame named `df` to retain only rows where the column `age` is greater than 30:

```
from pyspark.sql.functions import col

filtered_df = df.where(col("age") > 30)
```

Using **`filter`** *condition:*

You can also use the `filter()` function instead of the `where()` function, which works the same way.

```
from pyspark.sql.functions import col

filtered_df = df.filter(col("age") > 30)
```

> *Note that the `where()` function returns a new DataFrame with the filtered rows and the original DataFrame remains unchanged.*

2. Filtering rows with multiple conditions

In Apache Spark, you can use the `where()` function to filter rows in a DataFrame based on multiple conditions. You can chain multiple conditions together using the `&` (and) or `|` (or) operators.

Using **`where, '&'`** *on multiple conditions:*

For example, the following code filters a DataFrame named `df` to retain only rows where the column `age` is greater than 30 and the column `gender` is equal to "male":

```
from pyspark.sql.functions import col

filtered_df = df.where((col("age") > 30) & (col("gender") == "male"))
```

Using **`where & |`** *on multiple conditions:*

You can also use the `|` operator to filter rows where the column `age` is greater than 30 or the column `gender` is equal to "male"

```
from pyspark.sql.functions import col

filtered_df = df.where((col("age") > 30) | (col("gender") == "male"))
```

Using **`filter`** *on multiple conditions:*

You can also use the `filter()` function instead of the `where()` function, which works the same way.

```
from pyspark.sql.functions import col

filtered_df = df.filter((col("age") > 30) & (col("gender") == "male"))
```

Using **`where`** *on multiple conditions:*

It is also possible to use multiple conditions on the same column, like this:

```
from pyspark.sql.functions import col

filtered_df = df.where((col("age") > 30) & (col("age") < 40) & (col("gender"
```

You can chain as many conditions as you need and use different comparison operators such as `<`, `>`, `<=`, `>=`, `==`, `!=`, `like`, `rlike`, `between`, `in`, `isnull`, `isnotnull` etc.

> *Note that the `where()` function returns a new DataFrame with the filtered rows and the original DataFrame remains unchanged.*

2. Filtering on an Array column

3. Filtering on an Array column

In Apache Spark, you can use the `where()` function to filter rows in a DataFrame based on an array column. You can use the `array_contains()` function to check if a specific value exists in an array column.

Using **`where & array_contains`** *condition:*

For example, the following code filters a DataFrame named `df` to retain only rows where the column `colors` contains the value "red":

```
from pyspark.sql.functions import array_contains

filtered_df = df.where(array_contains(col("colors"), "red"))
```

Using **`filter & array_contains`** *condition:*

You can also use the `filter()` function instead of the `where()` function, which works the same way.

```
from pyspark.sql.functions import array_contains

filtered_df = df.filter(array_contains(col("colors"), "red"))
```

Using **`filter & size`** *condition:*

You can also use the `size` function to find the size of an array column, and filter on it

```
from pyspark.sql.functions import size

filtered_df = df.filter(size(col("colors")) > 3)
```

Using **`filter & array_except`** *condition:*

You can also use the `array_except` function to filter rows where a specific value is not in an array column

```
from pyspark.sql.functions import array_except

filtered_df = df.filter(array_except(col("colors"), "red").isNotNull())
```

> *Note that the `where()` and `filter()` functions return a new DataFrame with the filtered rows and the original DataFrame remains unchanged.*

It's also worth noting that you can use other array functions like `array_intersect`, `array_union` etc to perform filtering on array columns.

4. Filtering on Nested Struct columns

In Apache Spark, you can use the `where()` function to filter rows in a DataFrame based on a nested struct column. You can use the `.$fieldName` notation to access the fields of a struct column.

Using **`where`** *condition:*

For example, let's say you have a DataFrame named `df` with a struct column named `address` which has fields `city`, `state`, and `zipcode`. The following code filters the DataFrame to retain only rows where the `address.city` is equal to "New York":

```
filtered_df = df.where(col("address.city") == "New York")
```

Using **`filter`** *condition:*

You can also use the `filter()` function instead of the `where()` function, which works the same way.

```
filtered_df = df.filter(col("address.city") == "New York")
```

Using **`where`** *on multiple conditions:*

You can chain multiple conditions on nested struct columns, like this:

```
filtered_df = df.where((col("address.city") == "New York") & (col("address.z
```

Using **`where & getField`** *condition:*

You can also use the `getField()` function to access the field of a struct column

```
from pyspark.sql.functions import getField

filtered_df = df.where(getField(col("address"), "city") ==
```

Conclusion

In this article, I've explained how to filter rows from Spark DataFrame based on single or multiple conditions and SQL expressions using `where()` function. Alternatively, you also use `filter()` function to filter the rows on DataFrame.

Follow for more articles on PySpark. Do check out my previous articles on using PySpark for data testing. Happy reading..!!

Spark | QA | Data Engineering | Software Testing | Software Development

Written by Ahmed Uz Zaman

3K followers · 15 following

Follow

Lead QA Engineer | ETL Test Engineer | PySpark | SQL | AWS | Azure | Improvising Data Quality through innovative technologies | linkedin.com/in/ahmed-uz-zaman/

Responses (1)

A Aaditya Adhikari

What are your thoughts?

Siddharth Ghosh
Jul 30, 2023

Can you tell about the performance difference of both where and filter ? Which is better?

3 Reply

More from Ahmed Uz Zaman

Ahmed Uz Zaman

PySpark Date & Time Functions: A Comprehensive Guide

10 Essential PySpark Date and Time Functions for Data Analysis

Mar 16, 2023 109 2

In Geek Culture by Ahmed Uz Zaman

Pandas vs PySpark..!

Key differences, when to use either, free resources

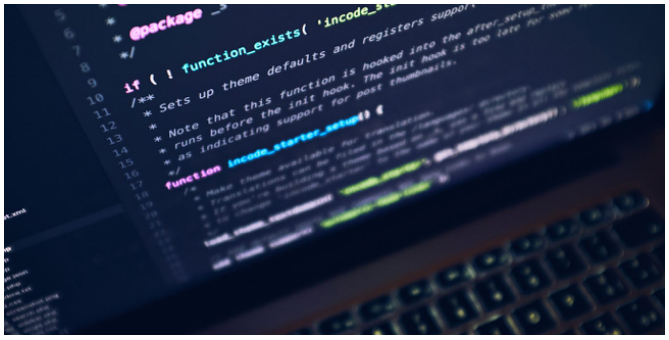
Jan 22, 2023 222 2

In Plumbers Of Data Scie... by Ahmed Uz Zam...

Exploring the Different Join Types in Spark SQL: A Step-by-Step...

Understand the Key Concepts and Syntax of Cross, Outer, Anti, Semi, and Self Joins

Feb 3, 2023 91 1



Ahmed Uz Zaman

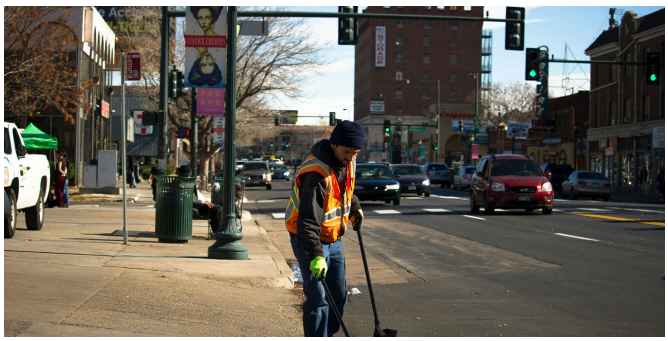
PySpark Window Functions: A Comprehensive Guide

Using Window Functions in PySpark: Examples and Explanations

Mar 18, 2023 97 1

See all from Ahmed Uz Zaman

Recommended from Medium



In Art of Data Engineering by Santosh Joshi

Understanding Collect, Take, Limit, Show, Head and Display i...

A Quick and Crisp Guide to Inspecting DataFrames Efficiently in PySpark

Jan 13 7

Mayurkumar Surani

🚀 Mastering PySpark: Your Complete Guide to 46 Essential...

Collection of PySpark Functions

Jun 3 36

Mukovhe Mukwevho

5 Ways to Check If a Spark DataFrame is Empty

Read here for free if you do not have a medium subscription!

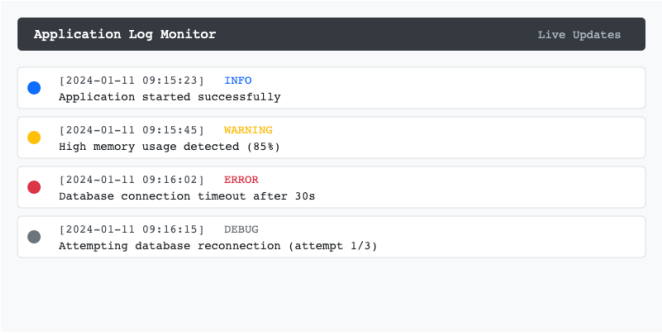
Feb 13 43 1

Siddharth Ghosh

Spark Interview Series—Different Optimization Techniques(RBO,...

Apache Spark has evolved into a powerful big data processing engine with several...

Apr 8

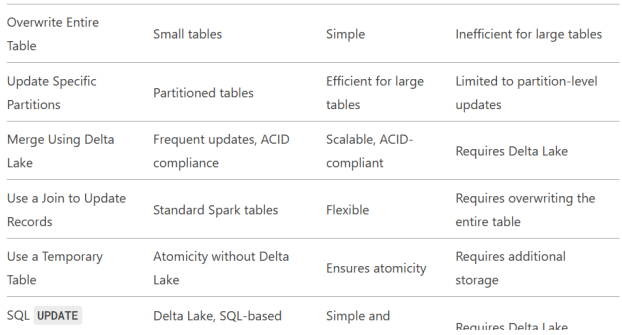


Ganesh Chandrasekaran

Databricks: Using Python logging module in notebooks

One common problem many developers face when using the Python logging librar...

Jan 12 30 2



Overwrite Entire Table	Small tables	Simple	Inefficient for large tables
Update Specific Partitions	Partitioned tables	Efficient for large tables	Limited to partition-level updates
Merge Using Delta Lake	Frequent updates, ACID compliance	Scalable, ACID-compliant	Requires Delta Lake
Use a Join to Update Records	Standard Spark tables	Flexible	Requires overwriting the entire table
Use a Temporary Table	Atomicity without Delta Lake	Ensures atomicity	Requires additional storage
SQL `UPDATE`	Delta Lake, SQL-based	Simple and	Requires Delta Lake

Sujatha Mudadla

Section 2: Data Processing—Updating Records in a Spark...

In data processing, Type 1 updates refer to overwriting existing records with new data...

Jan 11 1

See more recommendations