PySpark Where Filter Function | Multiple Conditions:

- PySpark **filter ( ) function** is used to **filter** the **rows** from **RDD/DataFrame** based on the given **condition** or **SQL** expression.
- You can also use **where ( ) clause** instead of the **filter ( )** if you are **coming** from **SQL** background, **both** these **functions operate exactly** the **same**.

1) DataFrame filter ( ) with Column Condition:

- Use **Column** with the **condition** to **filter** the **rows** from **DataFrame**.
- Using this you can **express complex condition** by **referring column names** using **dfObject.colname**

2) DataFrame filter ( ) with SQL Expression:

If you are **coming** from **SQL** background, you can **use** that **knowledge** in **PySpark** to **filter** DataFrame **rows** with **SQL** expressions.

3) PySpark Filter with Multiple Conditions:

- In PySpark, to **filter ( )** rows on **DataFrame** based on **multiple** conditions, you case use either **Column** with a **condition** or **SQL** expression.
- Below is **just** a **simple** example using **&** condition, you can **extend** this with **OR (|)**, and **NOT (!)** conditional expressions as needed.

#multiple condition:

4) Filter on an Array column:

When you want to **filter** rows from **DataFrame** based on **value** present in an **array collection** column you **can** use **array_contains ( )** from Pyspark SQL functions which **checks** if a **value contains** in an **array** if **present** it **returns true** otherwise **false.**

#Array Column Filter:

5) Filtering on Nested Struct columns:

If your **DataFrame** consists of **nested struct** columns, you can use any of the above syntaxes to filter the rows based on the nested column.

#Struct condition

PySpark orderby ( ), sort ( ) & groupBy ( ):

- You can **use** either **sort ( )** or **orderby ( ) functions** of PySpark **DataFrame** to **sort** DataFrame by **ascending** or **descending order** based on **single** or **multiple** columns.
- You **can** also do **sorting** using **PySpark SQL sorting functions**

1) DataFrame sorting using the sort ( ) & orderby ( ) function:

- PySpark **DataFrame class** provides **sort ( ) function** to **sort** on **one** or **more columns**.
- By **default**, it **sorts** by **ascending** order.

Note:

- The above **two** examples **return** the **same** output, the **first one** takes the **DataFrame column name** as a **string** and the **next** takes **columns** in **Column type**.
- This table **sorted** by the **first department column** and **then** the **state column**.

2) DataFrame sorting using orderby ( ) function:

PySpark DataFrame also provides **orderby ( ) function** to **sort** on **one** or **more** columns. By **default**, it **orders** by **ascending**.

3) Sort by Ascending (ASC):

If you want to specify the **ascending order/sort explicitly** on **DataFrame**, you can use the **asc method** of the **Column** function.

4) Sort by Descending (DESC):

If you want to specify the **sorting by descending order** on **DataFrame**, you can use the **desc method** of the **Column** function. for example.

2) groupBy ( ):

Similar to **SQL GROUP BY** clause, PySpark **groupBy ( ) function** is used to **collect** the **identical data** into **groups** on **DataFrame** and perform **aggregate functions** on the **grouped data**.

1) PySpark groupBy and aggregate on DataFrame columns:

Let's do the **groupBy ( )** on **department column** of DataFrame and then **find** the **sum** of **salary** for **each department** using **sum ( ) aggregate** function.

2) Similarly, we can calculate the number of employee in each department using count ( )

➔ **min, max, avg**

3) PySpark groupBy and aggregate on multiple columns:

Similarly, we can also run **groupBy** and **aggregate** on **two** or **more DataFrame columns**.

Let's group by on **department**, **state** and do **sum ( )** on **salary** and **bonus** columns.

#GroupBy on multiple columns:

4) Running more aggregates at a time:
Using **agg ( ) aggregate function** we can **calculate many aggregations** at a time **on a single** statement **using** PySpark SQL **aggregate functions sum ( ), avg ( ), min ( ), max ( ) mean ( )** etc. In **order** to **use** these, we should **import** from pyspark.sql.functions

5) Using filter on aggregate data:
Similar to SQL "**HAVING**" clause, On **PySpark DataFrame** we can **use** either **where ( )** or **filter ( )** function to **filter** the **rows** of **aggregated** data**.**

withColumn:
PySpark **withColumn ( )** is a **transformation function** of **DataFrame** which is **used** to **change** or **update** the **value**, **convert** the **datatype** of an **existing DataFrame column**, **add/create** a **new column**.

1) Change column DataType using PySpark withColumn:
- By using PySpark **withColumn ( )** on a **DataFrame**, we can **cast** or **change** the **data type** of a **column.**
- In **order** to **change data type**, you would also **need** to **use cast ( ) function** along with **withColumn ( ).**
- Below statement **changes** the **datatype** from **String** to **Integer** for "**salary**" **column.**

2) Update the value of an existing column:
- PySpark **withColumn ( ) function** of **DataFrame** can also be **used** to **change** the **value** of an **existing column.**
- In **order** to **change** the **value**, **pass** an **existing column name** as a **first argument** and **value** to be **assigned** as a **second argument** to the **withColumn ( )** function.

Note: The **second argument should** be of **Column type.**

3) Create a new column from an existing:
To **add/create** a **new** column, **specify** the **first argument** with a **name** you **want** your **new column to be** and **use** the **second argument** to **assign** a **value** by **applying** an **operation** on an **existing** column.

4) Add a new column using withColumn ( ):
- In order to **create** a **new column**, **pass** the **column name** you **want** to the **first argument** of **withColumn ( )** transformation function.
- Make sure this **new** column **is not present** on **DataFrame**, if it **presents** it **updates** the **value** of that **column.**
- On below snippet, **lit ( ) function** is **used** to **add** a **constant value** to a **DF column.**
- We **can** also **chain** in **order** to **add multiple columns.**

 # Add one column:

 # Add multiple columns:

5) Rename column name:

- Though you **cannot rename** a **column** using **withColumn**, still **renaming** is **one** of the **common operations** we **perform** on DataFrame.
- To **rename** an **existing column** use **withColumnRenamed** ( ) function on **DataFrame**.

6) Drop a column from PySpark DataFrame:

Use "**drop**" function to **drop** a **specific column** from the **DataFrame**.

withColumnRenamed:

- Use PySpark **withColumnRenamed ( )** to **rename** a **DataFrame column.**
- We often **need** to **rename one column** or **multiple columns** on PySpark **DataFrame**.

1) To rename single column name:

- PySpark has a **withColumnRenamed ( ) function** on **DataFrame** to **change** a **column** name.
- This **function** takes **two parameters**; the **first** is your **existing column name** and the **second** is the **new column name** you wish for.

2) To rename multiple columns:

To change **multiple column names**, we should **chain withColumnRenamed function.**

3) Using PySpark StructType: To rename a nested column in Dataframe:

Changing a **column name** on **nested data** is **not straight forward** and we **can do this** by **creating** a **new schema** with **new DataFrame columns** using **StructType** and **use** it **using cast function**.

Note:

This **statement renames firstname** to **fname** and **lastname** to **lname** within **name structure**.

#Select struct columns:

#Then drop the existing nested structure:

4) Using Select to rename nested elements:

PySpark Union and UnionAll:
PySpark **union ( )** and **unionAll ( )** transformations are **used** to **merge two** or **more DataFrame's** of the **same schema** or **structure**.

- **union ( ) method** of the **DataFrame** is used to **merge two DF** of the **same structure/schema**. If **schemas** are **not** the **same** it **returns** an **error**.
- **unionAll ( )** is **deprecated** since **Spark "2.0.0" version** and **replaced** with **union ( )**.

Note: In **SQL** languages, **Union eliminates** the **duplicates** but **UnionAll merges two datasets** including **duplicate** records.
But, in PySpark **both behave** the **same** and **recommend** using DF **duplicate ( )** function to **remove duplicate** rows.

Second DataFrame:
Now, let's **create** a **second Dataframe** with the **new records** and **some records** from the **above Dataframe** but with the **same schema**.

1) Merge two or more DataFrames using union:
DF **union ( )** method **merges two DF** and **returns** the **new DF** with **all rows** from **two DF regardless** of **duplicate** data.

2) Merge DataFrames using unionAll:
DF **unionAll ( )** method is **deprecated** since **Spark "2.0.0" version** and **recommends using** the **union ( )** method.

Note: **Returns** the **same output** as **above**.

3) Merge without Duplicates:
Since the **union ( )** method **returns all rows without distinct records**, we will **use** the **distinct ( )** function to **return** just **one record** when **duplicate exists**.

drop – dropDuplicates:
PySpark **distinct ( )** function is **used** to **drop** the **duplicate rows** (**all columns**) from **DF** and **dropDuplicates ( )** is **used** to **drop** selected (**one or multiple**) columns.

1) Get distinct all columns:
Above **DF**, we have a **total** of **10 rows** with **2 rows** having **all values duplicated**, **performing distinct** on this **DF** should get us **9 rows**.

Alternatively, you can also run **dropDuplicates ( )** function which **return** a **new** DF with **duplicate rows removed**.

2) PySpark Distinct of multiple columns:

- PySpark **doesn't** have a **distinct method** which **takes columns** that should **run distinct** on (**drop duplicate rows on selected columns**)
- However, it **provides another signature** of **dropDuplicates ( )** function which takes **multiple columns** to **eliminate duplicates.**

Note: Calling **dropDuplicates ( )** on DF **returns** a **new DataFrame** with **duplicate rows removed**.

3) Drop columns:

#Single Column:

#Multiple Columns:

case – when – others:

- In PySpark DataFrame, "**when otherwise**" is used **derive** a **column** or **update** an **existing column** based on **some conditions** from **existing columns** data.
- **when ( ) is a SQL function** with a **return** type **Column** and **other ( )** is a **function** in **sql.Column class.**

Like **SQL** "**case when**" statement and "Switch", "**if then else**" statement also supports similar syntax using "**when otherwise**" or using "**case when**" statement.

1) Using "when otherwise" on PySpark DataFrame:
when( ) is a PySpark **SQL function**, so to **use** it **first** we should **import**
from pyspark.sql.functions import when.

Let's **replace** the **value** of **gender** with new **derived value**, **when value not qualified** with the **condition**, we are **assigning** "**Unknown**" as **value.**

2) Using "case when" on PySpark DataFrame:
Similarly, we could use "**case when**" with expression **expr ( )** and **withColumn ( ).**

#case with select:

3) Using & and | operator:
We can also use **and (&) or (|)** within **when** function. Let's create a **new set of data** to **make** it **simple**.

String Functions:
pyspark.sql.functions provides **two** functions **concat ( )** and **concat_ws ( )** to **concatenate** DF **multiple** columns into a **single** column.

1) PySpark concatenate using concat ( ):
- **concat ( )** function of **Pyspark SQL** is used to **concatenate multiple DF columns** into a **single column**.

Syntax: pyspark.sql.functions.concat(*cols)

2) PySpark concat_ws ( ) Usage:
concat_ws ( ) function of Pyspark **concatenates multiple string columns** into a **single column** with a given **separator** or **delimiter**.

Syntax: pyspark.sql.functions.concat_ws(**sep**,*cols)

Date/Time Functions:
In PySpark, you can do **almost all** the **date operations** you can think of using **in-built functions**.

Create a dataframe with sample date values:

Now the **problem** I see here is that columns **start**_dt & **end**_dt are of **type string** and **not date**. So let's **quickly convert** it into **date**.

Now we are **good**. We have a **DF** with 2 columns **start_dt** & **end_dt**. Both the **columns** are of datatype '**date**'. Let's do some Date operations on this.

1) Change Date Format:

2) Fetch Current Date:

3) Add Days to date:

4) Subtract days from date:

5) Subtract 2 dates:

6) Add Months to date:

7) Add Years to date:

8) Extract Year, Month, Day, WeekofYear, DayofWeek, DayofYear from Date:

9) Last Day of Month:

10) Determine how many months between 2 Dates:

11) Identify Next Day:
Monday:
Tuesday:

11) Fetch quarter of the year:

12) Truncate Date to Year, Month:

Aggregate Functions:

- PySpark provides **built-in** standard **Aggregate functions** defined in **DF.**
- **Aggregate** functions **operate** on a **group** of **rows** and **calculate** a **single return value** for **every group**.
- All these **aggregate functions** accept input as **column type** or **column name** in a **string** and **several other arguments** based on the **function** and **return** column type.
- **Aggregate functions** are little bit more **compile-time safety**, **handles null** and **perform better** when **compared** to **UDF's**.
- If your **application** is **critical** on **performance** try to **avoid** using **custom UDF**.
- UDF's **does not guarantee** on **performance**.

1) approx_count_distinct:
approx_count_distinct ( ) function **returns** the **count** of **distinct items** in a **group**.

2) avg (average):
avg( ) function returns the average of values in the input column.

3) collect_list:
collect_list ( ) function **returns all values** from an **input column** with **duplicates**.

4) collect_set:
collect_set ( ) function **returns all values** from an **input column** with **NO duplicate values**.

5) countDistinct:
countDistinct ( ) function **returns** the **number** of **distinct elements** in a **columns**.

6) count function:
count ( ) function **returns number** of **elements** in a **column**.

7) first/last function:
first() function returns the first/last element in a column when ignoreNulls is set to true, it returns the first non-null element.

9) sumDistinct function:
sumDistinct ( ) function **returns** the **sum** of **all distinct values** in a **column**.

Window Functions:

- PySpark **Window** functions **operate** on a **group** of **rows** (**like frame, partition**) and **return** a **single value** for **every input row**.
- To **perform** an **operation** on a **group first** we **need** to **partition** the **data using Window.partitionBy ( )**
- For **row number** and **rank function** we **need** to **additionally order by** on **partition data** using **orderBy** clause.

1) ranking functions
2) analytic functions
3) aggregate functions

1) row_number:
row_number ( ) window function is **used** to **give** the **sequential row number** starting **from 1** to the **result** of each **window partition**.

2) rank:
rank( ) window function is **used** to **provide** a **rank** to the **result within** a **window partition**. This function **leaves gaps** in **rank** when **there** are **ties**.

3) dense_rank:

- **dense_rank ( )** window function is **used** to get the **result** with **rank** of **rows within** a **window partition without any gaps**.
- This is similar to **rank ( )** function **difference** being **rank** function **leaves gaps** in rank **when there** are **ties** but **dense_rank does not leave gaps**.

4) ntile:
ntile ( ) window function **returns** the **relative rank** of **result rows within** a **window partition**.
If we **provide 2** as an **argument** to **ntile** it **returns ranking between 2 values** (1 and 2).

5) lag:
This is the same as the **LAG** function in **SQL**.

6) lead:
This is the same as the **LEAD** function in **SQL.**

7) Window Aggregate Functions:
- Let's see how to **calculate sum**, **min**, **max** for **each department** using **PySpark SQL Aggregate window functions** and **WindowSpec**.
- When **working** with **Aggregate functions** we **don't need** to **use order by** clause.

8) Explode Function:
- PySpark **explodes array** and **map columns** to **rows**.
- **PySpark function explode (e: Column)** is **used** to **explode** or **create array** or **map columns** to **rows.**
- When an **array** is **passed** to this **function**, it **creates** a **new default column** "**col1**" and it **contains all array** elements.
- When a **map** is **passed**, it **creates two new columns one for key** and **one** for **value** and **each element** in **map split into** the **rows.**

Note:
This will **ignore elements** that have **null** or **empty.**
Wilma and **Jatin** have **null** or **empty values** in **array** and **map** hence the following snippet **does not contain** these **rows.**

1) explode array:

2) explode map:

Joins:
- **Joins** is **used** to **combine two DF.**
- It **supports all** basic **join operations available** in **traditional SQL.**
- **Spark Joins** are **wider transformations** that **involve data shuffling across** the **network.**
- **Spark SQL Joins** comes with **more optimization** by **default** (thanks to DataFrames).

Syntax: join (self, other, on=None, how=None)

- **join** operation **takes parameters** as **below** and **returns DataFrame.**
- **other Right side** of the **join**
- **on** a **string** for the **join column name**
- **how default inner.**

Must be **one** of **inner**, **cross**, **outer**, **full**, **full_outer**, **left**, **left_outer**, **right**, **right_outer**, **left_semi** and **left_anti**.

1) Inner Join:

- **Inner join** is the **default join** in PySpark and it's **mostly used**.
- This **joins two datasets** on **key** columns where **keys don't match** the **rows get dropped** from **both datasets** (emp & dept).

Note: When we apply **Inner join** on our **datasets**, it **drops** "**emp_dept_id**" **60** from "**emp**" and "**dept_id**" **30** from "**dept**" **datasets**.

2) Full Outer Join:

Outer a.k.a full, **fullouter** join **returns all rows** from **both datasets** where **join expression doesn't match** it **returns null** on **respective record columns**.

Note: From our "**emp**" dataset's "**emp_dept_id**" with **value 60 doesn't have** a **record** on "**dept**" hence **dept columns have null** and "**dept_id**" **30 doesn't have** a **record** in "**emp**" hence **you see null's on emp columns.**

3) Left Outer Join:

- **Left a.k.a. Leftouter** join **returns all rows** from the **left dataset** regardless of **match found** on the **right dataset.**
- When **join expression doesn't match**, it **assigns null** for that **record** and **drops records** from **right** where **match not found**.

Note: From our dataset, "**emp_dept_id**" **60 doesn't have** a **record** on "**dept**" dataset hence, this record **contains null** on "**dept**" columns (**dept_name & dept_id**) and "**dept_id**" **30** from "**dept**" dataset **dropped** from the **results**.

4) Right Outer Join:

- **Right a.k.a Rightouter join** is **opposite** of **left join**, here it **returns all rows** from the **right dataset regardless** of **match found** on the **left dataset**.
- When **join** expression **doesn't match**, it **assigns null** for that **record** and **drops records** from **left** where **match not found**.

Note: From our example, the right dataset "**dept_id**" **30 doesn't have** it on the **left dataset** "**emp**" hence, this **record contains null** on "**emp**" columns and "**emp_dept_id**" **60 dropped** as a **match not found** on **left**.

4) Left Semi Join:

- **leftsemi join** is **similar** to **inner join difference** being **leftsemi** join **returns all** columns from the **left** dataset and **ignores all columns** from the **right** dataset.
- In other words, this **join returns** columns from the **only left** dataset for the **records match** in the **right dataset on join expression** records **not matched** on **join expression** are **ignored** from **both left and right** datasets.
- The **same result** can be **achieved** using **select** on the **result** of the **inner join** however using this **join** would be **efficient**.

5) Left Anti Join:

- **leftanti join** does the **exact opposite** of the **leftsemi**.
- **leftanti** join **returns only columns** from the **left dataset** for **non-matched records**.

6) PySpark Self Join:

- **Joins** are **not complete without** a **self-join**.
- Though there is **no self-join type** available we **can use** any of the **above-explained join types** to **join DF** to **itself**.

Note: Here, we are joining **emp** dataset **with itself** to find out **superior emp_id** and **name** for **all employees**.

7) Using SQL Expression:

Since **PySpark SQL support native SQL** syntax we can also **write join operations** after **creating temporary tables** on DF and use these tables on **spark.sql ( )**.

8) PySpark SQL Join on multiple DataFrame's:

When you **need to join** more than **two tables** you either use **SQL expression** after **creating** a **temporary view** on the **DF** or **use** the **result** of **join operation** to **join** with **another DF** like **chaining them**.