Pyspark UDFs: What, when, and how to use it?

Pallavi Sinha Follow 6 min read · Apr 28, 2023

Source: https://spark.apache.org/images/

UDF (User-defined function) in PySpark is a feature that can be used to extend its functionality. UDFs enable us to perform complex data processing tasks by creating our own functions in Python and applying them to Spark data frames.

In this blog, we will go through a basic understanding of what is UDF in

Pyspark, when can we use it, and how to use it (Best practices). So, let's dive into it -

What is UDF?

> *PySpark UDF is a User defined function that once created, can be used for multiple data frames. UDFs can be used to perform various transformations on Spark dataframes, such as data cleaning, parsing, aggregation, and more. They are invoked for every row in a dataset.*

Let us take a simple example. Suppose we have a dataframe with a column- `number` .

```
df = spark.createDataFrame([6,7,3,8,2,9],"int").toDF("number")
df.show()
```

```
Original dataframe -
+------+
|number|
+------+
|     6|
|     7|
|     3|
|     8|
|     2|
|     9|
+------+
```

We want to add a column `is_even` . This column will contain the value "yes" if the corresponding `number` column value is an even number else "no". We don't have any PySpark in-built SQL function to do this job directly. Hence we can create a UDF for this and reuse it in as many dataframes as needed. Let us try to implement this example to learn creating UDFs.....

Step — 1: Write the custom logic that you want to implement as a UDF

```
def even_or_odd(num : int):
    if num % 2 == 0:
        return "yes"
    return "no"
```

We created a Python function that takes a number and returns "yes" if the number is even else it returns “no”.

Step — 2: Convert the Python function to PySpark UDF

```
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType

even_or_odd_udf = udf(lambda x : even_or_odd(x), StringType())
```

Here we converted our custom Python function to UDF by passing it to the PySpark SQL function - `udf()` . The default return type of `udf()` is `StringType()` .

Step — 3: Using UDF with data frame

```
print("Dataframe after adding is_even column - ")
result_df = df.withColumn("is_even", even_or_odd_udf(df["number"]))
result_df.show()
```

```
Dataframe after adding is_even column -
+------+-------+
|number|is_even|
+------+-------+
|     6|    yes|
|     7|     no|
|     3|     no|
|     8|    yes|
|     2|    yes|
|     9|     no|
+------+-------+
```

Here we used `even_or_odd_udf()` as a regular PySpark built-in function on `number` column of the dataframe to get the result.

This way we can use UDF to implement any custom logic which cannot be found in built-in PySpark SQL functions. We can use `even_or_odd()` with PySpark SQL query as well. For that, we need to register this function with PySpark. Below is the code for this —

```
spark.udf.register("even_or_odd_udf", even_or_odd , StringType())
df.createOrReplaceTempView("numbers_table") # created a temporary view from
spark.sql("select number, even_or_odd_udf(number) as is_even from numbers_ta
     .show()
```

It will yield the same result dataframe as above. We can create UDF in one step using annotation as well. Below is the code for it —

```
@udf(return_type = StringType())
def even_or_odd(num : int):
    if num % 2 == 0:
        return "yes"
    return "no"
```

After getting a clear understanding of what is UDF and what are the ways to create and use it, let us move forward to know when to use UDFs....

When to use UDF?

You must have heard somewhere that UDF is slow in PySpark. So let's try to compare the time taken by a UDF and a PySpark in-built function for the same use case through an example —

Suppose we have a dataframe with columns : `roll_no`, `first_name` and `last_name`. We want to concatenate `first_name` and `last_name` columns to create a `name` column. As we know that in PySpark we already have a `concat()` function and we don't require to write a UDF for this, but just for the sake of comparison let's do it both ways.

```
df = spark.createDataFrame([(1, "Ram", "Sharma"), (2, "Sita", "Kumari")], ['
df.show()
```

```
+-------+----------+---------+
|roll_no|first_name|last_name|
+-------+----------+---------+
|      1|       Ram|   Sharma|
|      2|      Sita|   Kumari|
+-------+----------+---------+
```

Option — 1: Using UDF —

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
import timeit

start = timeit.default_timer()

concat_udf = udf(lambda x,y : x + " " + y, StringType())
std_df = df.select("roll_no", concat_udf(df["first_name"], df["last_name"]).
std_df.show()

stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
+-------+-----------+
|roll_no|       name|
+-------+-----------+
|      1| Ram Sharma|
|      2|Sita Kumari|
+-------+-----------+

Time:  0.5883491250000006
```

Option — 2: Using PySpark SQL function —

```
from pyspark.sql.functions import concat, lit
import timeit
start = timeit.default_timer()

std_df = df.select("roll_no", concat(df["first_name"],lit(" "), df["last_nam
std_df.show()
stop = timeit.default_timer()
print('Time: ', stop - start)
```

```
+-------+-----------+
|roll_no|       name|
+-------+-----------+
|      1| Ram Sharma|
|      2|Sita Kumari|
+-------+-----------+

Time:  0.13245862500000172
```

We can clearly see that the time taken by UDF is more than the PySpark SQL function `concat()` . The difference will increase significantly in real-world scenarios on larger data and more manipulations. Let us see the reason behind it…

UDF is slow?

Implementing UDF should not be the first option. This is mainly due to the performance implications of UDFs such as :

1. UDFs are considered as a **black box** by catalyst optimizer in Spark and therefore cannot be optimized by Spark. All optimizations such as predicate pushdown, and constant folding are lost.
2. Spark engine is implemented in Java and Scala languages that run on JVM. Use of Python APIs (e.g. for UDFs) requires interaction between JVM and Python runtime. So, there is an additional overhead in using UDFs in PySpark, because the structures native to the JVM environment that Spark runs in, have to be converted to Python data

structures to be passed to UDFs, and then the results of UDFs have to be converted back. This happens via the py4j library which allows us to call code from JVM. This **back-and-forth serialization and deserialization** between JVM and Python runtime is a costly operation.

> *To summarize, we must use Spark SQL built-in functions as these functions provide optimizations. Consider creating UDFs only when the existing built-in SQL function doesn't have it.*

After understanding when to use UDFs, let us see what are the best practices we should consider while using UDFs…

What are the best practices for UDFs?

1. UDFs should be designed to handle null values correctly. Suppose, we have a column with some null records on it and we are passing it to the UDF, it will throw an `Atrribute error`. To avoid this, it is always best practice to check the null inside a UDF function. In any case, if we can't do a null check inside UDF, we must use conditional statements to check for null and call UDF conditionally.

2. We should define the input and output schema of UDFs explicitly. This will help reduce the risk of errors and improve performance.

3. Keep UDFs simple and specific for the tasks. This will help in making testing easier and easier maintenance.

Conclusion :

Therefore, we can say that UDF is a very useful way to bring our custom

logic to the PySpark dataframes and because of its performance implications we must use it only when we don't have a choice in PySpark in-built SQL functions.

Hope you enjoyed reading this blog. Feel free to ask questions and add your feedback in the comments. Thank you!

Pyspark Data Engineering Spark Python Pyspark Dataframes

Written by Pallavi Sinha

339 followers · 10 following

Follow

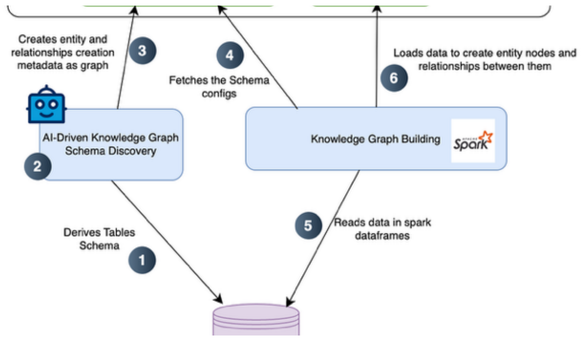
Data and AI enthusiast

No responses yet

A Aaditya Adhikari

What are your thoughts?

More from Pallavi Sinha

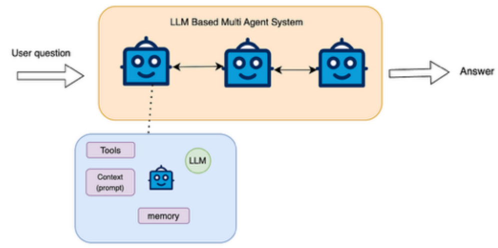


Pallavi Sinha

AI-Driven Knowledge Graph Schema Discovery: Concept an…

A knowledge graph structures data into entities (things like people, product, or…

Mar 7 244 5

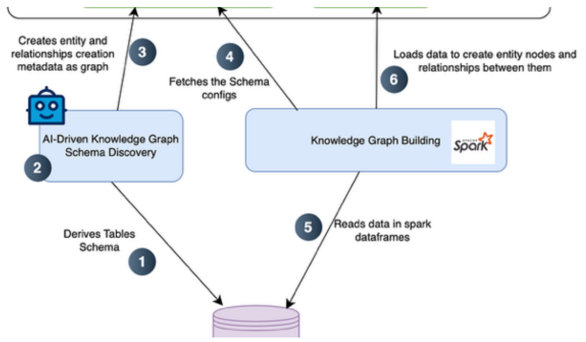


Pallavi Sinha

Understanding LLM-Based Agents and their Multi-Agent…

In today's era, large language models (LLMs) have emerged as formidable…

May 1, 2024 189

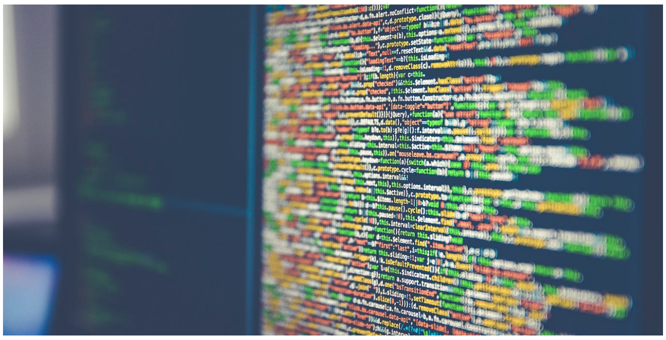


Pallavi Sinha

Building Knowledge Graph in Neo4j using PySpark

In our previous blog, we explored the core concepts of a Knowledge Graph—what it i…

Mar 7 12



Pallavi Sinha

Create Data Quality Framework with Great Expectations

In the modern era of data-driven decision-making, data quality has emerged as a…

Aug 7, 2023 132 4

See all from Pallavi Sinha

Recommended from Medium

In Towards Data Engineeri... by Ritam Mukherj...

Understanding Spark's Catalyst Optimizer: Demystifying Query...

Non-members can access the full article through this Link.

6d ago 10

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

Mayuran

Databricks LakeBase: Is This the Future of OLTP on the...

Understanding LakeBase: Databricks' new solution for real-time, low-latency data...

Jun 14 · 17 · 2

Victor Oketch Sabare

Stop Learning Pandas: The Spark 4.0 Features Nobody's Telling Y...

Apache Spark 4.0 finally lets Python-first data folks skip the "collect-to-Pandas"...

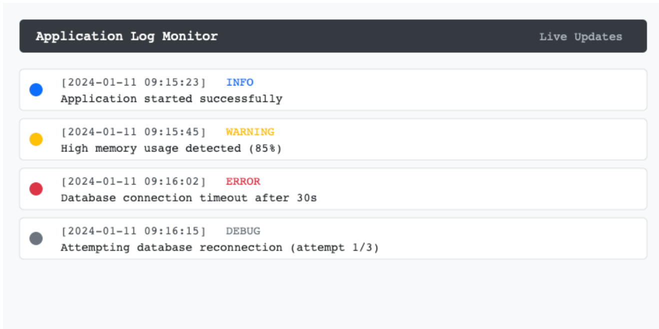
Jun 10 · 38 · 2

PY In Python in Plain English by Siddharth Ghosh

How Can You Optimize Spark SQL Queries?—An Interview Guide f...

Whether you're getting ready for a big data engineering interview or trying to improve...

Jun 14



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

See more recommendations