

What is the purpose of StructType and StructField in PySpark, and how can they be used in data structures? Provide examples.

Authored by
stats writer

June 24, 2024

RECOMMENDED CITATION

stats writer (2024). *What is the purpose of StructType and StructField in PySpark, and how can they be used in data structures? Provide examples..* PSYCHOLOGICAL SCALES.

Retrieved from <https://scales.arabpsychology.com/?p=150641>

StructType and StructField are two important components in PySpark that are used for defining the structure and schema of a data structure. The purpose of using these components is to provide a consistent and organized way of storing and accessing data in PySpark.

StructType is a data type that represents a collection of fields or columns in a dataset. It is similar to a table or a spreadsheet in traditional databases. StructType is used to define the structure of a DataFrame in PySpark, which is a distributed collection of data organized into named columns.

StructField, on the other hand, represents a single field or column in a StructType. It contains the data type and the name of the field. StructField is used to define the properties of each column in a StructType, such as the data type, nullability, and metadata.

These two components are used together to create a well-defined structure for data in PySpark. They can be used in various data structures, such as DataFrames, Datasets, and RDDs, to provide a consistent and structured way of storing and manipulating data.

For example, let's say we have a dataset containing information about students, including their name, age, and grade. We can define the structure of this dataset using StructType and StructField as follows:

```
StructType()
```

This defines a StructType with three fields: name, age, and grade, each with their respective data types. This structure can then be used to create a DataFrame in PySpark, which can be further manipulated and analyzed using various PySpark operations.

In conclusion, StructType and StructField are essential components in PySpark that are used for defining the structure and schema of data structures. They provide a consistent and organized way of storing and accessing data, making it easier to work with large datasets in PySpark.

The StructType and StructField classes in PySpark are used to specify the custom schema to the DataFrame and create complex columns like nested struct, array, and map columns. StructType is a collection of StructField objects that define column name, column data type, boolean to specify if the field can be nullable or not, and metadata.

In this article, I will explain different ways to define the structure of DataFrame using StructType with PySpark examples.

Key Points:

Related Articles:

1. StructType - Defines the structure of the DataFrame

PySpark provides `StructType` class from `pyspark.sql.types` to define the structure of the DataFrame.

`StructType` represents a schema, which is a collection of `StructField` objects. A `StructType` is essentially a list of fields, each with a name and data type, defining the structure of the DataFrame. It allows for the creation of nested structures and complex data types.

`PySpark printSchema()` method on the DataFrame shows `StructType` columns as `struct`.

2. StructField - Defines the metadata of the DataFrame column

It represents a field in the schema, containing metadata such as the name, data type, and nullable status of the field. Each `StructField` object defines a single column in the DataFrame, specifying its name and the type of data it holds.

The `StructField` class is also part of `pyspark.sql.types`

3. Using PySpark StructType & StructField with DataFrame

Regardless of how you create a DataFrame, you have the option to specify the custom schema using the `StructType` and `StructField` classes. As highlighted earlier, `StructType` is a collection of `StructFields`; with `StructType` you have an option to specify the column names, data types, and whether they're nullable.

With `StructField`, you can even incorporate nested struct schemas, `ArrayType` for arrays, and `MapType` for key-value pairs.

The following example demonstrates a basic example for creating a `StructType` and `StructField` for a DataFrame, along with sample data.

```
# Imports
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

spark = SparkSession.builder.master("local")
    .appName('SparkByExamples.com')
    .getOrCreate()
```

```
data =  
  
schema = StructType()  
  
df = spark.createDataFrame(data=data,schema=schema)  
df.printSchema()  
df.show(truncate=False)
```

By running the above snippet, it displays the outputs below.

```
# Output  
root  
|-- firstname: string (nullable = true)  
|-- middlename: string (nullable = true)  
|-- lastname: string (nullable = true)  
|-- id: string (nullable = true)  
|-- gender: string (nullable = true)  
|-- salary: integer (nullable = true)  
  
+-----+-----+-----+-----+-----+-----+  
|firstname|middlename|lastname|id |gender|salary|  
+-----+-----+-----+-----+-----+-----+  
|James | |Smith |36636|M |3000 |  
|Michael |Rose | |40288|M |4000 |  
|Robert | |Williams|42114|M |4000 |  
|Maria |Anne |Jones |39192|F |4000 |  
|Jen |Mary |Brown | |F |-1 |  
+-----+-----+-----+-----+-----+-----+
```

4. Defining Nested StructType object struct

To define a nested StructType in PySpark, use inner StructTypes within StructFields. Each nested StructType is a collection of StructFields, forming a hierarchical structure for representing complex data within DataFrames.

In the example below, the "name" column is of data type StructType, indicating a nested structure. This means that the "name" column itself is composed of multiple subfields or attributes, forming a hierarchical structure within the DataFrame.

```
# Defining schema using nested StructType
structureData =
structureSchema = StructType(),
StructField('id', StringType(), True),
StructField('gender', StringType(), True),
StructField('salary', IntegerType(), True)
])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)
```

This yields the below schema and the DataFrame

```
# Output
root
|-- name: struct (nullable = true)
| |-- firstname: string (nullable = true)
| |-- middlename: string (nullable = true)
| |-- lastname: string (nullable = true)
|-- id: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: integer (nullable = true)
```

```
+-----+-----+-----+
|name |id |gender|salary|
+-----+-----+-----+
| |36636|M |3100 |
| |40288|M |4300 |
||42114|M |1400 |
||39192|F |5500 |
| |F |-1 |
+-----+-----+-----+
```

5. Adding & Changing struct of the DataFrame

Using [PySpark SQL function struct\(\)](#), we can change the struct of the existing DataFrame and add a new StructType to it. The below example demonstrates how to copy the columns from one structure to another and adding a new column. [PySpark Column Class](#) also provides some functions to work with the StructType column.

```
# Updating existing structtype using struct
from pyspark.sql.functions import col,struct,when
updatedDF = df2.withColumn("OtherInfo",
struct(col("id").alias("identifier"),
col("gender").alias("gender"),
col("salary").alias("salary"),
when(col("salary").cast(IntegerType()) < 2000,"Low")
.when(col("salary").cast(IntegerType()) < 4000,"Medium")
.otherwise(High)).alias("Salary_Grade")
)).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)
```

Here, it copies "gender", "salary" and "id" to the new struct "otherInfo" and add's a new column "Salary_Grade".

```
# Output
root
|-- name: struct (nullable = true)
| |-- firstname: string (nullable = true)
| |-- middlename: string (nullable = true)
| |-- lastname: string (nullable = true)
|-- OtherInfo: struct (nullable = false)
| |-- identifier: string (nullable = true)
| |-- gender: string (nullable = true)
| |-- salary: integer (nullable = true)
| |-- Salary_Grade: string (nullable = false)
```

6. Using SQL ArrayType and MapType

SQL StructType also supports ArrayType and MapType to define the DataFrame columns for array and map collections, respectively. In the below example, column `hobbies` defined as `ArrayType(StringType)` and `properties` defined as `MapType(StringType,StringType)` meaning both key and value as String.

```
# Using SQL ArrayType and MapType
arrayStructureSchema = StructType(),
StructField('hobbies', ArrayType(StringType()), True),
StructField('properties', MapType(StringType(),StringType()), True)
```

```
] )
```

Outputs the below schema. Note that field `Hobbies` is array type and `properties` is map type.

```
# Output
root
|-- name: struct (nullable = true)
|   |-- firstname: string (nullable = true)
|   |-- middlename: string (nullable = true)
|   |-- lastname: string (nullable = true)
|-- hobbies: array (nullable = true)
|   |-- element: string (containsNull = true)
|-- properties: map (nullable = true)
|   |-- key: string
|   |-- value: string (valueContainsNull = true)
```

7. Creating StructType object struct from JSON file

Alternatively, you can load the SQL StructType schema from JSON file. To make it simple, I will get the current DataFrame schemas using `df2.schema.json()`, store this in a file, and use it to create a schema from this JSON file.

```
# Using json() to load StructType
print(df2.schema.json())
```

Yields below output.

```
{
  "type" : "struct",
  "fields" :
  },
  "nullable" : true,
  "metadata" : { }
}, {
  "name" : "dob",
  "type" : "string",
  "nullable" : true,
  "metadata" : { }
```

```
}, {  
"name" : "gender",  
"type" : "string",  
"nullable" : true,  
"metadata" : { }  
}, {  
"name" : "salary",  
"type" : "integer",  
"nullable" : true,  
"metadata" : { }  
} ]  
}
```

Alternatively, you could also use `df.schema.simpleString()`, this will return a relatively simpler schema format.

Now, let's load the JSON file and use it to create a DataFrame.

```
# Loading json schema to create DataFrame  
import json  
schemaFromJson = StructType.fromJson(json.loads(schema.json))  
df3 = spark.createDataFrame(  
spark.sparkContext.parallelize(structureData), schemaFromJson)  
df3.printSchema()
```

8. Creating StructType object struct from DDL String

To create a StructType object, 'struct', from a Data Definition Language (DDL) string in PySpark, use 'StructType.fromDDL()'. This method parses the DDL string and generates a StructType object that reflects the schema defined in the string.

For example, 'struct = StructType.fromDDL("name STRING, age INT")' creates a StructType with two fields: 'name' of type 'STRING' and 'age' of type 'INT'. This allows for dynamic schema creation based on DDL specifications, facilitating seamless integration with external systems or data sources where schema information is defined using DDL.

```
# Create StructType from DDL String  
ddlSchemaStr = "`fullName` STRUCT<`first`: STRING, `last`: STRING,  
`middle`: STRING>, `age` INT, `gender` STRING"  
ddlSchema = StructType.fromDDL(ddlSchemaStr)
```

```
ddlSchema.printTreeString()
```

9. Check DataFrame Column Exists

To check if a column exists in a PySpark DataFrame, use the 'contains()' method on the DataFrame's 'columns' attribute. For example, 'if "column_name" in df.columns' checks if the column exists in DataFrame 'df'. Alternatively, you can use 'selectExpr()' with the column name and 'alias()' to create a new column with a different name, then check if the new column exists.

```
# Check if a column exists
if "firstname" in df.columns:
    print("Column 'firstname' exists in the DataFrame.")
else:
    print("Column 'firstname' does not exist in the DataFrame.")
```

Alternatively, you can also use.

```
# Checking Column exists using contains()
print(df.schema.fieldNames.contains("firstname"))
print(df.schema.contains(StructField("firstname", StringType, true)))
```

This example returns "true" for both scenarios.

10. Complete Example of PySpark StructType & StructField

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, ArrayType, MapType
from pyspark.sql.functions import col, struct, when
```

```
spark = SparkSession.builder.master("local")
    .appName('SparkByExamples.com')
    .getOrCreate()
```

```
data =
```

```
schema = StructType()
```

```
df = spark.createDataFrame(data=data,schema=schema)
df.printSchema()
df.show(truncate=False)

structureData =
structureSchema = StructType(),
StructField('id', StringType(), True),
StructField('gender', StringType(), True),
StructField('salary', IntegerType(), True)
])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)
df2.printSchema()
df2.show(truncate=False)

updatedDF = df2.withColumn("OtherInfo",
struct(col("id").alias("identifier"),
col("gender").alias("gender"),
col("salary").alias("salary"),
when(col("salary").cast(IntegerType()) < 2000,"Low")
.when(col("salary").cast(IntegerType()) < 4000,"Medium")
.otherwise(High).alias("Salary_Grade")
)).drop("id","gender","salary")

updatedDF.printSchema()
updatedDF.show(truncate=False)

""" Array & Map """

arrayStructureSchema = StructType(),
StructField('hobbies', ArrayType(StringType()), True),
StructField('properties', MapType(StringType(),StringType()), True)
])
```

Find a complete example at the [GitHub project](#).

Conclusion:

In conclusion, PySpark's StructType and StructField classes offer powerful tools for defining and managing DataFrame schemas. StructType provides a structured approach to organizing data, enabling precise specification of column names, data types, and nullability. Meanwhile, StructField

allows for fine-grained control over individual fields within the schema, including nested structures and complex data types.

By leveraging PySpark's StructType and StructField, users can effectively structure and manipulate data within DataFrames, ensuring consistency and enabling seamless integration with diverse data processing workflows.

Happy Learning !!

Related Articles

ARABPSYCHOLOGY.COM