Raushan Kumar

PYSPARK

JSON FUNCTIONS:

- from_json()
- to_json()
- get_json_object()
- json_tuple()
- schema_of_json()

Raushan Kumar
https://www.linkedin.com/in/raushan-kumar-553154297/

PYSPARK JSON FUNCTIONS

In PySpark, there are a variety of functions available for working with JSON data. These functions allow you to read, write, and manipulate JSON data effectively in your DataFrames.

Here’s a list of JSON-related functions in PySpark

1), from_json()

The **from_json()** function in **PySpark** is used to parse a **JSON string** column into structured data types like **StructType** or **MapType** according to a **schema** that you define. It is particularly useful when you have JSON data stored as strings in a column and need to convert it into a format that's easier to process, query, and manipulate in PySpark.

Syntax

```
from pyspark.sql.functions import from_json

from pyspark.sql.types import StructType, StructField, StringType,
IntegerType

from_json(col: Column, schema: StructType or DataType)
```

- col: The column containing the JSON string to be parsed.
- schema: The schema that defines the structure of the JSON data (using StructType and StructField).

Real-Time Example of from_json()

Imagine a real-world scenario where you have a log file or data stream that contains user information in JSON format. You want to process this JSON data and extract meaningful insights using PySpark.

Example Scenario:

You are working with a DataFrame that contains a column of JSON strings representing user activity data. Each JSON string contains user information, activity details, and timestamp.

Here is a sample of the data in the user_activity_json column:

```
{

  "user_id": "user_001",

  "name": "John Doe",

  "activity": "Login",

  "timestamp": "2025-02-18T15:32:00"

}
```

Objective:

- Parse the JSON string into structured data (i.e., user_id, name, activity, timestamp).
- Convert the **timestamp** into a proper **DateType** format.
- Create a DataFrame that is easier to query and analyze.

Step-by-Step Process:

1. **Define the Schema:**
   - Define the schema for the JSON data. This helps PySpark understand how to interpret the JSON fields and map them to corresponding types (e.g., StringType, TimestampType).
2. **Apply the from_json() Function:**
   - Use the from_json() function to parse the JSON strings into structured data.
3. **Transform the Data:**
   - Convert the parsed data into a more useful format (e.g., extracting timestamps, or additional transformations).

PySpark Code Implementation:

1), Define Schema

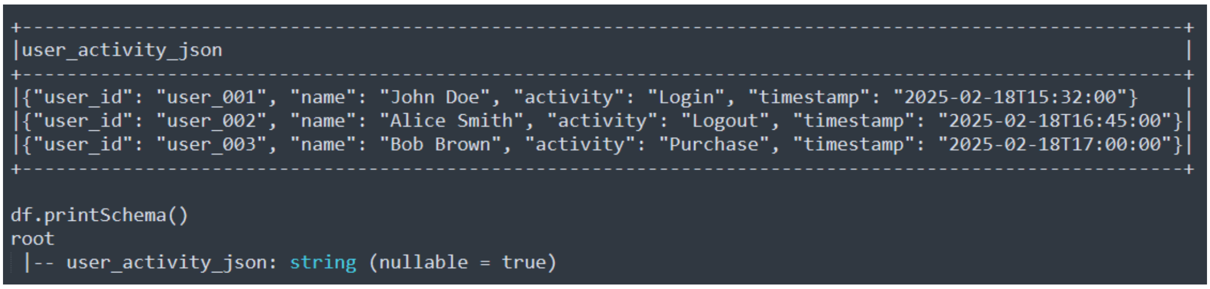
Let's define a schema to parse the JSON data.

```
from pyspark.sql.types import StructType, StructField, StringType,
TimestampType

# Define the schema for the JSON string
schema = StructType([
   StructField("user_id", StringType(), True),
   StructField("name", StringType(), True),
   StructField("activity", StringType(), True),
   StructField("timestamp", StringType(), True)  # Initially as
StringType, will be converted later
])
```

2), Sample DataFrame with JSON Strings

Create a DataFrame with JSON string data in a column.

```
from pyspark.sql import SparkSession
# Initialize Spark session
spark = SparkSession.builder.appName("from_json-
example").getOrCreate()
# Sample Data with JSON string
data = [
   ('{"user_id": "user_001", "name": "John Doe", "activity": "Login",
"timestamp": "2025-02-18T15:32:00"}',),
   ('{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout",
"timestamp": "2025-02-18T16:45:00"}',),
   ('{"user_id": "user_003", "name": "Bob Brown", "activity":
"Purchase", "timestamp": "2025-02-18T17:00:00"}',)
]
# Create DataFrame with JSON column
df = spark.createDataFrame(data, ["user_activity_json"])
df.show(truncate=False)
```

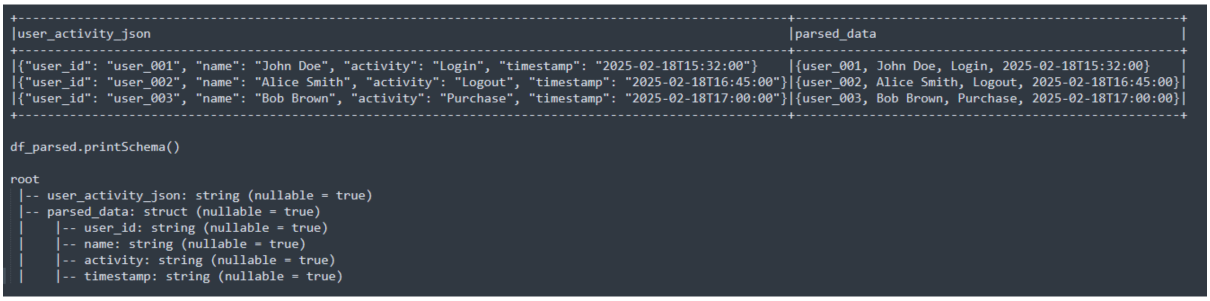


```
+--------------------------------------------------------------------------------------------------------+
|user_activity_json                                                                                      |
+--------------------------------------------------------------------------------------------------------+
|{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}    |
|{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}|
|{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}|
+--------------------------------------------------------------------------------------------------------+

df.printSchema()
root
 |-- user_activity_json: string (nullable = true)
```

3), Parse the JSON Data Using from_json()

We’ll use the from_json() function to convert the **user_activity_json** column from a JSON string into a structured format based on the schema we defined.

```
from pyspark.sql.functions import from_json

# Parse the JSON string column using the defined schema

df_parsed = df.withColumn("parsed_data", from_json("user_activity_json", schema))

df_parsed.show(truncate=False)
```



```
+--------------------------------------------------------------------------------------------------------+---------------------------------------------------+
|user_activity_json                                                                                      |parsed_data                                        |
+--------------------------------------------------------------------------------------------------------+---------------------------------------------------+
|{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}    |{user_001, John Doe, Login, 2025-02-18T15:32:00}    |
|{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}|{user_002, Alice Smith, Logout, 2025-02-18T16:45:00}|
|{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}|{user_003, Bob Brown, Purchase, 2025-02-18T17:00:00}|
+--------------------------------------------------------------------------------------------------------+---------------------------------------------------+

df_parsed.printSchema()

root
 |-- user_activity_json: string (nullable = true)
 |-- parsed_data: struct (nullable = true)
 |    |-- user_id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- activity: string (nullable = true)
 |    |-- timestamp: string (nullable = true)
```

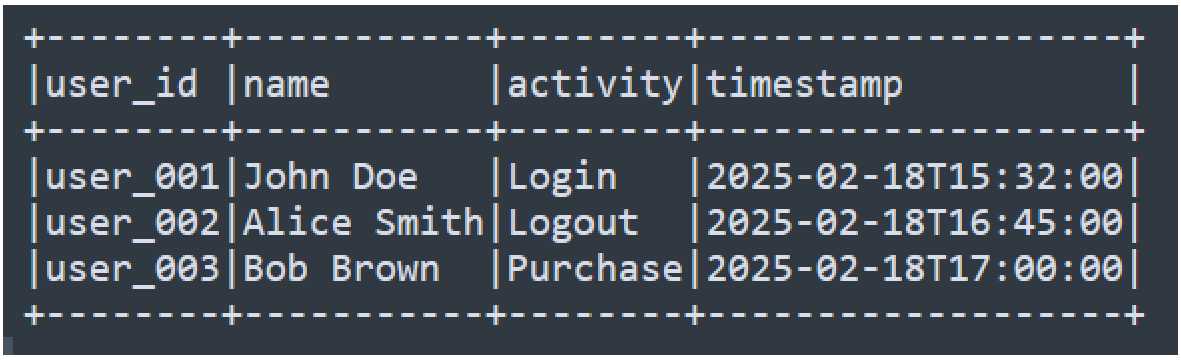
Notice that the **user_activity_json** column has been parsed into the **parsed_data** column, which is now a struct containing user_id, name, activity, and timestamp.

4), Extract Individual Fields from Parsed Data

You can now extract individual fields from the **parsed_data** column to make it easier to query and analyze.

```
df_final = df_parsed.select(
  "parsed_data.user_id",
  "parsed_data.name",
  "parsed_data.activity",
  "parsed_data.timestamp"
)
df_final.show(truncate=False)
```

Output



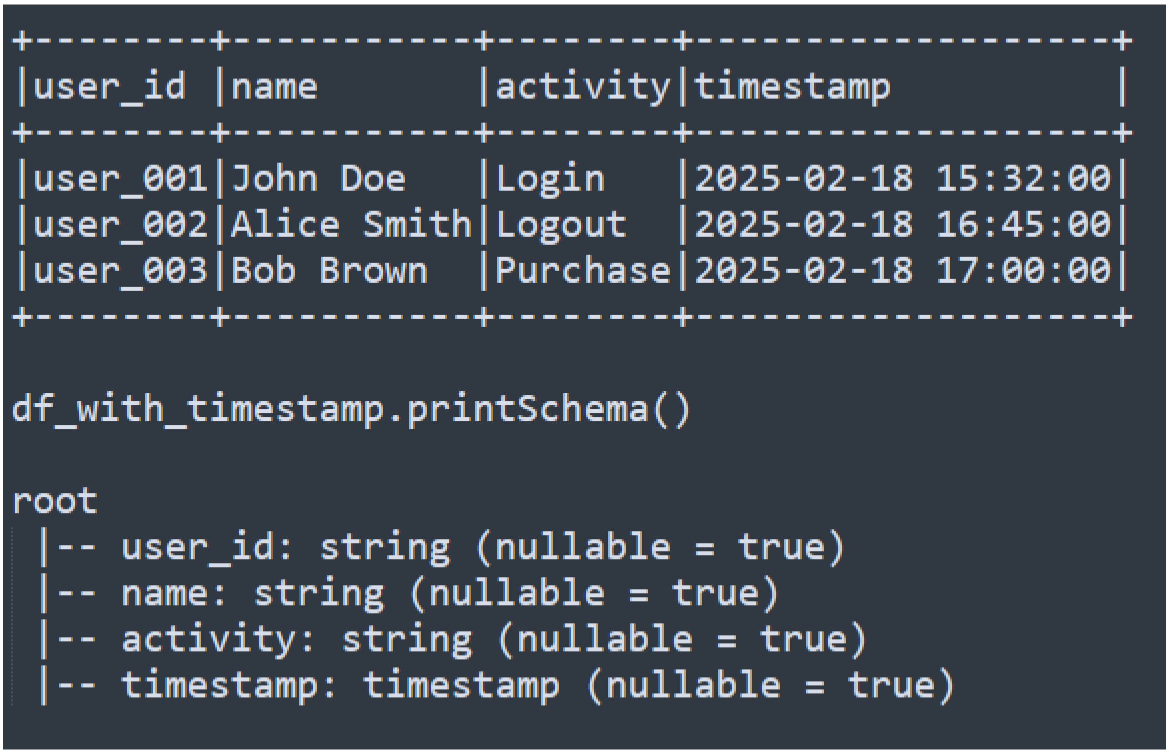
```
+--------+-----------+--------+-------------------+
|user_id |name       |activity|timestamp          |
+--------+-----------+--------+-------------------+
|user_001|John Doe   |Login   |2025-02-18T15:32:00|
|user_002|Alice Smith|Logout  |2025-02-18T16:45:00|
|user_003|Bob Brown  |Purchase|2025-02-18T17:00:00|
+--------+-----------+--------+-------------------+
```

5), Convert timestamp to TimestampType

You can convert the timestamp field to an actual **TimestampType** to perform time-based operations like filtering, aggregation, etc.

```
from pyspark.sql.functions import col, to_timestamp
# Convert timestamp to TimestampType for proper datetime operations
df_with_timestamp = df_final.withColumn("timestamp",
to_timestamp("timestamp", "yyyy-MM-dd'T'HH:mm:ss"))
df_with_timestamp.show(truncate=False)
```

Output



```
+--------+-----------+--------+-------------------+
|user_id |name       |activity|timestamp          |
+--------+-----------+--------+-------------------+
|user_001|John Doe   |Login   |2025-02-18 15:32:00|
|user_002|Alice Smith|Logout  |2025-02-18 16:45:00|
|user_003|Bob Brown  |Purchase|2025-02-18 17:00:00|
+--------+-----------+--------+-------------------+

df_with_timestamp.printSchema()

root
 |-- user_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- activity: string (nullable = true)
 |-- timestamp: timestamp (nullable = true)
```

Key Takeaways:

- **from_json()** is useful when working with JSON data that is stored as strings in a column. It allows you to parse the JSON into structured types like StructType and MapType.
- You need to define a **schema** that matches the structure of the JSON data in order to correctly parse it.
- After parsing the JSON data, you can extract fields and manipulate them further (e.g., converting strings to timestamps, etc.).

2), to_json():

The to_json() function in **PySpark** is used to convert **structured data types** like StructType, ArrayType, or MapType into a **JSON string**.

This function is particularly useful when you need to store or transmit your data in JSON format after performing transformations on a **DataFrame**.

to_json() Function:

Syntax

```
from pyspark.sql.functions import to_json

to_json(col: Column)
```

col: The column containing **structured data** (like StructType, ArrayType, or MapType) that you want to convert into a **JSON string**.

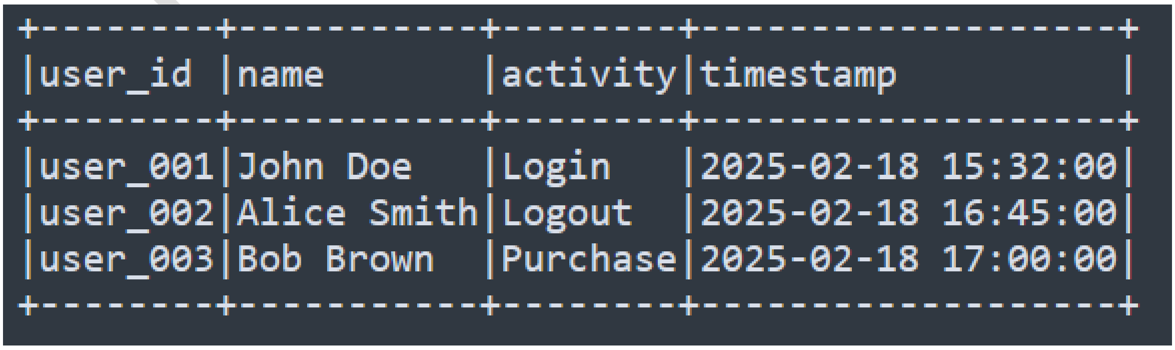
Real-Time Example of to_json()

Imagine a real-world scenario where you have structured data representing **user activity logs**. After performing some transformations and aggregations on the data, you want to output the results in **JSON format** to store them in a file or send them over a network.

Example Scenario:

You are working with a **DataFrame** that contains **user information** and **activity details**. The data is in a structured format, and you want to **serialize** it into a **JSON string** for downstream processes.

Here is the structure of your DataFrame before applying the to_json() function:



```
+--------+-----------+--------+-------------------+
|user_id |name       |activity|timestamp          |
+--------+-----------+--------+-------------------+
|user_001|John Doe   |Login   |2025-02-18 15:32:00|
|user_002|Alice Smith|Logout  |2025-02-18 16:45:00|
|user_003|Bob Brown  |Purchase|2025-02-18 17:00:00|
+--------+-----------+--------+-------------------+
```

Objective:

- Convert the structured data into a **JSON string** format.
- Output the results as a JSON string for each record.

Step-by-Step Process:

1. **Create a Sample DataFrame** with structured data (StructType).
2. **Apply the to_json() Function** to convert structured data into a JSON string.
3. **View the Results** with JSON strings.

PySpark Code Implementation:

1), Create a Sample DataFrame

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
# Initialize Spark session
spark = SparkSession.builder.appName("to_json-example").getOrCreate()
# Sample data
data = [
  ('user_001', 'John Doe', 'Login', '2025-02-18 15:32:00'),
  ('user_002', 'Alice Smith', 'Logout', '2025-02-18 16:45:00'),
  ('user_003', 'Bob Brown', 'Purchase', '2025-02-18 17:00:00')
]
# Define schema
schema = StructType([
  StructField("user_id", StringType(), True),
  StructField("name", StringType(), True),
  StructField("activity", StringType(), True),
  StructField("timestamp", StringType(), True)])
```

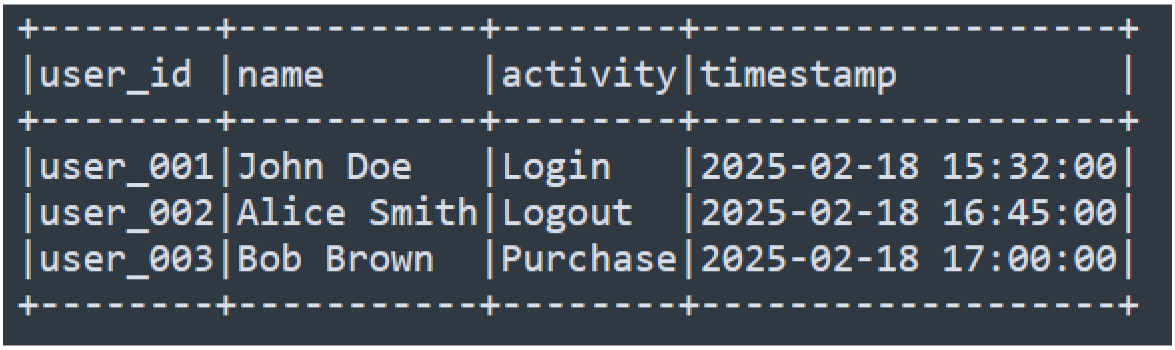
# Create DataFrame

df = spark.createDataFrame(data, schema)

# Show the DataFrame

df.show(truncate=False)

Output



```
+--------+-----------+--------+-------------------+
|user_id |name       |activity|timestamp          |
+--------+-----------+--------+-------------------+
|user_001|John Doe   |Login   |2025-02-18 15:32:00|
|user_002|Alice Smith|Logout  |2025-02-18 16:45:00|
|user_003|Bob Brown  |Purchase|2025-02-18 17:00:00|
+--------+-----------+--------+-------------------+
```

2. Apply to_json() to Convert Structured Data to JSON String

Now, let’s apply the **to_json()** function to the DataFrame. We will convert the entire row into a JSON string.

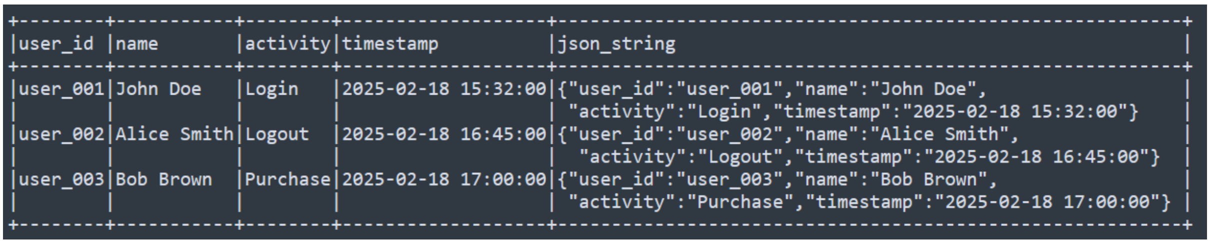
from pyspark.sql.functions import to_json, struct

# Use struct to combine columns into a struct, then apply to_json to convert to JSON string

df_json = df.withColumn("json_string", to_json(struct("user_id", "name", "activity", "timestamp")))

# Show the resulting DataFrame with JSON string

df_json.show(truncate=False)



```
+--------+-----------+--------+-------------------+----------------------------------------------------------+
|user_id |name       |activity|timestamp          |json_string                                               |
+--------+-----------+--------+-------------------+----------------------------------------------------------+
|user_001|John Doe   |Login   |2025-02-18 15:32:00|{"user_id":"user_001","name":"John Doe",                  |
|        |           |        |                   | "activity":"Login","timestamp":"2025-02-18 15:32:00"}    |
|user_002|Alice Smith|Logout  |2025-02-18 16:45:00|{"user_id":"user_002","name":"Alice Smith",               |
|        |           |        |                   |  "activity":"Logout","timestamp":"2025-02-18 16:45:00"}  |
|user_003|Bob Brown  |Purchase|2025-02-18 17:00:00|{"user_id":"user_003","name":"Bob Brown",                 |
|        |           |        |                   | "activity":"Purchase","timestamp":"2025-02-18 17:00:00"} |
+--------+-----------+--------+-------------------+----------------------------------------------------------+
```

Key Takeaways:

- The **to_json()** function converts structured data (such as StructType, ArrayType, or MapType) into a **JSON string**.
- It can handle **nested data** structures, turning them into properly formatted JSON strings.
- The function is useful when you need to **serialize** your data for storage, transmission, or integration with other systems that consume JSON.

This functionality is widely used in scenarios like **data serialization**, **APIs**, **log data**, or **data interchange between systems**.

3. get_json_object():

The get_json_object() function in **PySpark** is used to extract a **specific field** from a **JSON string** based on a **JSON path**. It allows you to retrieve data from a JSON string or **JSON column** by specifying a JSON path expression.

Syntax

```
from pyspark.sql.functions import get_json_object

get_json_object(col: Column, path: String)
```

- col: The column containing the **JSON string**.
- path: The **JSON path expression** to access the required field from the JSON string.

Real-Time Example of get_json_object()

Imagine you're working with a real-world scenario where you have a **log file** or a **stream of data** containing **JSON strings**, and you need to extract specific fields from those JSON strings. Let's take an example where user activity data is stored in JSON format in a column. Your goal is to extract specific fields, such as **user ID**, **name**, and **activity**.

Example Scenario:

You have a DataFrame with a column user_activity_json containing user activity information in **JSON format**. You want to extract specific fields like user_id, activity, and timestamp.

Here's a sample of the **JSON string** stored in the user_activity_json column:

```
{
  "user_id": "user_001",
  "name": "John Doe",
  "activity": "Login",
  "timestamp": "2025-02-18T15:32:00"
}
```

Objective:

Extract the values of user_id, activity, and timestamp from the JSON string using the get_json_object() function.

Step-by-Step Process:

1. **Create a Sample DataFrame** with JSON strings.
2. **Apply get_json_object()** to extract specific fields from the JSON string.
3. **View the Results** with the extracted fields.

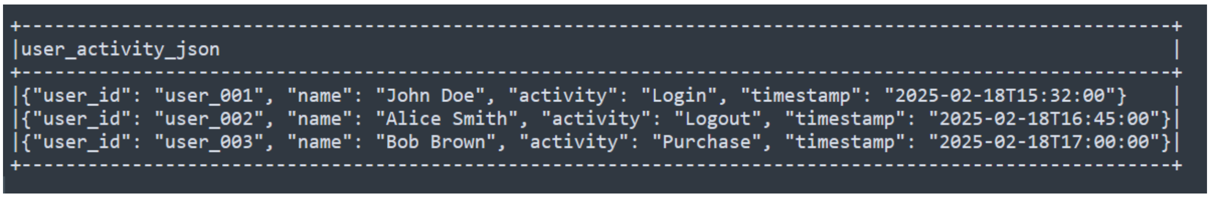
PySpark Code Implementation:

1. Create a Sample DataFrame

Let’s create a DataFrame with a column user_activity_json that contains JSON strings.

```
from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.appName("get_json_object-example").getOrCreate()

# Sample data with JSON strings

data = [

    ('{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}',),

    ('{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}',),

    ('{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}',)

]

# Create DataFrame with JSON string column

df = spark.createDataFrame(data, ["user_activity_json"])

# Show the DataFrame

df.show(truncate=False)
```

Output



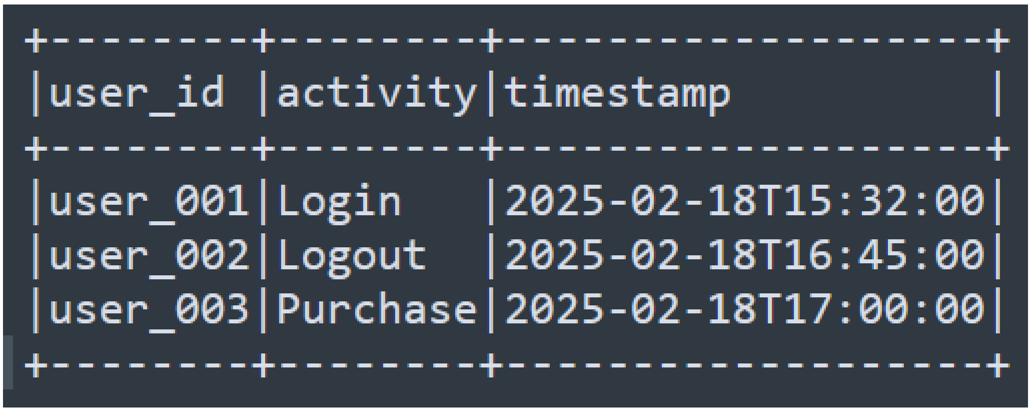
```
+--------------------------------------------------------------------------------------------------------+
|user_activity_json                                                                                      |
+--------------------------------------------------------------------------------------------------------+
|{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}    |
|{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}|
|{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}|
+--------------------------------------------------------------------------------------------------------+
```

2. Apply get_json_object() to Extract Fields

Now, let's use the **get_json_object()** function to extract specific fields like user_id, activity, and timestamp from the JSON string in the user_activity_json column.

```
from pyspark.sql.functions import get_json_object

# Extract specific fields from the JSON string using get_json_object

df_extracted = df.select(

  get_json_object("user_activity_json", "$.user_id").alias("user_id"),

  get_json_object("user_activity_json", "$.activity").alias("activity"),

  get_json_object("user_activity_json", "$.timestamp").alias("timestamp")

)

# Show the DataFrame with the extracted fields

df_extracted.show(truncate=False)
```

Output



```
+--------+--------+-------------------+
|user_id |activity|timestamp          |
+--------+--------+-------------------+
|user_001|Login   |2025-02-18T15:32:00|
|user_002|Logout  |2025-02-18T16:45:00|
|user_003|Purchase|2025-02-18T17:00:00|
+--------+--------+-------------------+
```

Explanation:

- get_json_object("user_activity_json", "$.user_id"): This extracts the **user_id** field from the **JSON string** in the user_activity_json column.
- get_json_object("user_activity_json", "$.activity"): This extracts the **activity** field.

- get_json_object("user_activity_json", "$.timestamp"): This extracts the **timestamp** field.

The $.field_name syntax is a **JSON path** expression that allows you to access fields from the JSON object.

3. Extracting Nested Fields

In case the JSON structure is nested, you can use get_json_object() to extract nested fields.

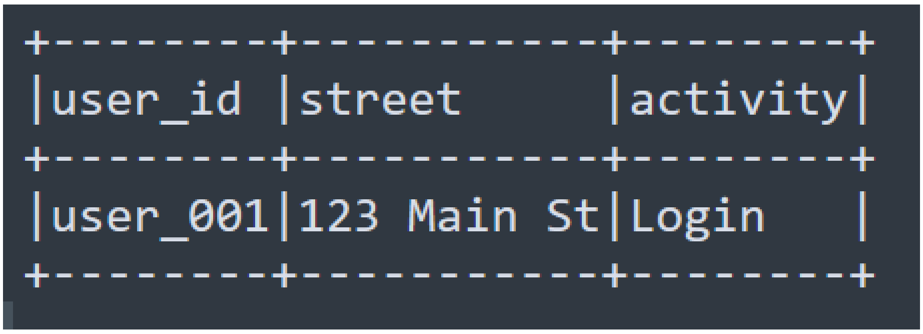
Let's assume the JSON structure looks like this:

```
{
 "user_id": "user_001",
 "name": "John Doe",
 "address": {
  "street": "123 Main St",
  "city": "New York"
 },
 "activity": "Login",
 "timestamp": "2025-02-18T15:32:00"
}
```

To extract the **street** field from the **address** object, you would use:

```
# Extract nested fields using get_json_object
df_nested_extracted = df.select(
  get_json_object("user_activity_json", "$.user_id").alias("user_id"),
  get_json_object("user_activity_json", "$.address.street").alias("street"),
  get_json_object("user_activity_json", "$.activity").alias("activity")
)
df_nested_extracted.show(truncate=False)
```

Output



```
+--------+-----------+--------+
|user_id |street     |activity|
+--------+-----------+--------+
|user_001|123 Main St|Login   |
+--------+-----------+--------+
```

Here, we used the **JSON path** $.address.street to extract the street field from the **nested address object**.

Real-World Use Case: Extracting User Data

Let's consider a real-world use case where you have user data in a **JSON log file**. Each log contains a user's activity along with their metadata. You want to analyze the activities for each user, extracting specific fields such as **user ID**, **activity type**, and **timestamp**.

This can be useful in systems that generate **JSON logs** (like web logs, server logs, or IoT device logs) and you want to extract certain attributes for further analysis.

For example:

- **Web application logs** might have JSON fields like user_id, session_id, page, activity, etc.
- **IoT device logs** might have JSON fields like device_id, temperature, status, etc.

Using get_json_object() will allow you to process large volumes of JSON data efficiently.

Key Takeaways:

- **get_json_object()** is used to extract specific fields from a **JSON string** based on a **JSON path**.
- The function is useful when you have **JSON data** in a column and you want to retrieve specific fields, especially when dealing with complex or nested JSON objects.
- The **JSON path** uses $.field_name syntax, and you can use it for **nested fields**.
- This function is widely used in real-time data processing scenarios, such as parsing **logs** or **event data** stored in JSON format.

4), json_tuple()

The json_tuple() function in PySpark is used to extract multiple values from a JSON string column and return them as separate columns. It is an efficient way to extract multiple fields from a JSON object in one operation.

Syntax:

```
from pyspark.sql.functions import json_tuple

json_tuple(col: Column, *fields: String)
```

- col: The column containing the **JSON string**.
- fields: The list of **fields** you want to extract from the JSON string. You can specify multiple fields to extract in a single call.
- json_tuple() is often used when you have a **JSON column** with several fields and you want to **extract multiple fields** from that JSON object at once.

Real-Time Example of json_tuple()

Let's consider a **real-world scenario** where you are working with a dataset containing **JSON strings**. The JSON data may contain **user activity logs** and you need to extract several fields (such as **user_id**, **activity**, **timestamp**, etc.) from that JSON string.

For example, a column called user_activity_json in a DataFrame contains JSON strings like:

```
{
  "user_id": "user_001",
  "name": "John Doe",
  "activity": "Login",
  "timestamp": "2025-02-18T15:32:00"
}
```

You can use json_tuple() to **extract** user_id, activity, and timestamp into separate columns in your DataFrame.

Example Scenario:

You have a **DataFrame** with a column user_activity_json that contains JSON data. Your task is to **extract specific fields** (like **user_id**, **activity**, **timestamp**) from the JSON string.

Step-by-Step Process:

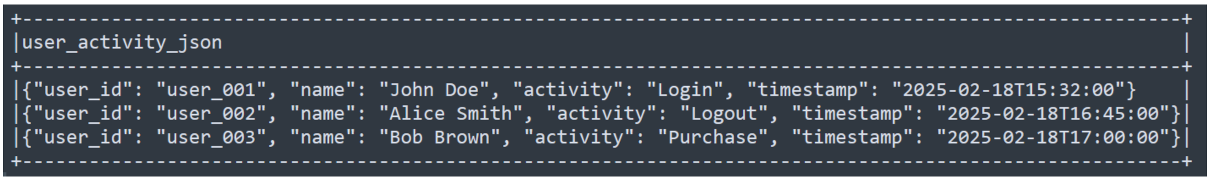
1. **Create a Sample DataFrame** with JSON strings.
2. **Apply json_tuple()** to extract multiple fields.
3. **View the Results** with the extracted fields.

PySpark Code Implementation:

1. Create a Sample DataFrame

Let’s create a DataFrame with a column user_activity_json that contains **JSON strings**.

```
from pyspark.sql import SparkSession

# Initialize Spark session

spark = SparkSession.builder.appName("json_tuple-example").getOrCreate()

# Sample data with JSON strings

data = [

    ('{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}',),

    ('{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}',),

    ('{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}',)

]

# Create DataFrame with JSON string column

df = spark.createDataFrame(data, ["user_activity_json"])

# Show the DataFrame

df.show(truncate=False)
```



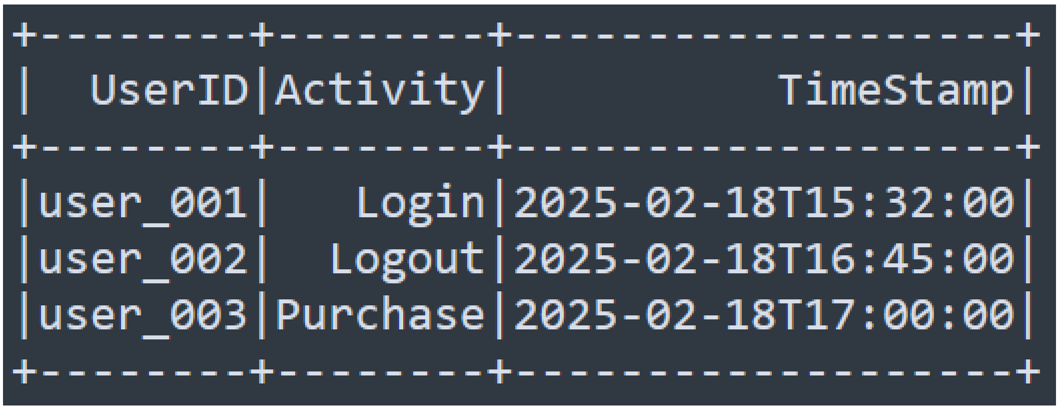
```
+--------------------------------------------------------------------------------------------------------+
|user_activity_json                                                                                      |
+--------------------------------------------------------------------------------------------------------+
|{"user_id": "user_001", "name": "John Doe", "activity": "Login", "timestamp": "2025-02-18T15:32:00"}    |
|{"user_id": "user_002", "name": "Alice Smith", "activity": "Logout", "timestamp": "2025-02-18T16:45:00"}|
|{"user_id": "user_003", "name": "Bob Brown", "activity": "Purchase", "timestamp": "2025-02-18T17:00:00"}|
+--------------------------------------------------------------------------------------------------------+
```

2. Apply json_tuple() to Extract Fields

We will now use the json_tuple() function to extract specific fields from the user_activity_json column and return them as separate columns.

```
from pyspark.sql.functions import json_tuple,col

df_extracted=df.select(json_tuple(col('user_activity_json'),'user_id','activity','timestamp').alias('UserID','Activity','TimeStamp'))

df_extracted.show()
```

Output



```
+--------+--------+-------------------+
|  UserID|Activity|          TimeStamp|
+--------+--------+-------------------+
|user_001|   Login|2025-02-18T15:32:00|
|user_002|  Logout|2025-02-18T16:45:00|
|user_003|Purchase|2025-02-18T17:00:00|
+--------+--------+-------------------+
```

Here, the json_tuple() function extracts the **user_id**, **activity**, and **timestamp** fields from the **JSON string** and creates new columns in the DataFrame.

3. Extracting Nested Fields (Example)

If your JSON data has **nested fields**, json_tuple() can still be used for **simple** extraction of the top-level fields, but it does not support deep nesting. For example, suppose you have the following JSON structure:

```
{
 "user_id": "user_001",
 "name": "John Doe",
 "address": {
  "street": "123 Main St",
  "city": "New York"
 },
 "activity": "Login",
 "timestamp": "2025-02-18T15:32:00"
}
```

To extract top-level fields (e.g., user_id, activity, timestamp), you can still use json_tuple():

```
# Extract top-level fields using json_tuple

df_nested_extracted = df.select(json_tuple("user_activity_json", "user_id", "activity", "timestamp").alias("user_id", "activity", "timestamp"))

df_nested_extracted.show(truncate=False)
```

However, if you need to extract nested fields like street and city, you would need to use a combination of functions such as get_json_object() or selectExpr().

json_tuple() will only work with top-level fields and is not suited for deeply nested structures.

Example: Handling Deeply Nested JSON

To extract the street field from the nested address object, you can use get_json_object() as follows:

# Extract nested field using get_json_object

```
df_nested_extracted = df.select(

   get_json_object("user_activity_json", "$.user_id").alias("user_id"),

   get_json_object("user_activity_json", "$.address.street").alias("street"),

   get_json_object("user_activity_json", "$.activity").alias("activity")

)

df_nested_extracted.show(truncate=False)
```

Key Takeaways:

- **json_tuple()** is used to extract **multiple fields** from a **JSON string** column in PySpark.
- It is very efficient for extracting **multiple top-level fields** from a JSON object at once.
- The function returns each extracted field as a **separate column**.

- json_tuple() does **not support nested fields**, so you should use other functions like **get_json_object()** for deeper extractions.
- It is ideal for cases where you have **JSON logs** or **structured data** stored in a JSON format, and you need to extract multiple fields for further analysis.

5), schema_of_json()

The schema_of_json() function in PySpark is used to infer the schema of a JSON string column. This function is particularly useful when working with unstructured or semi-structured data (such as JSON files) because it automatically infers the structure of the JSON data, which can be used for further operations, such as parsing or querying.

Syntax

```
from pyspark.sql.functions import schema_of_json
```

```
schema_of_json(jsonString: Column)
```

- jsonString: A Column containing a JSON string. This column must be of the StringType, and schema_of_json() will infer the schema of the JSON string.

The function returns the schema of the JSON string in the form of a StructType. This is useful when you have a JSON column and you need to know its structure (fields and their types).

Real-Time Example:

Let's look at a **real-world example** where we have a dataset with **JSON strings**, and we want to **infer the schema** of the JSON data.

Example Scenario:

You have a column in a **DataFrame** that contains **JSON strings** representing **user information**. The data is semi-structured, and you want to infer its schema to understand its structure.

For example, a JSON string might look like this:

```
{
  "user_id": "user_001",
  "name": "John Doe",
  "address": {
    "street": "123 Main St",
    "city": "New York"
  },
  "is_active": true,
  "last_login": "2025-02-18T15:32:00"
}
```

You want to infer the schema from this JSON data and use it for further analysis.

Step-by-Step Example:

1. **Create a DataFrame** with a column containing JSON strings.
2. **Use schema_of_json()** to infer the schema of the JSON string.
3. **Display the inferred schema** and use it for further processing.

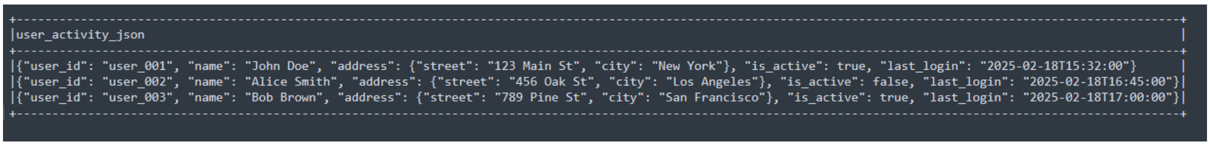
PySpark Code Implementation:

1. Create a Sample DataFrame

Let’s first create a **DataFrame** containing a column user_activity_json with JSON strings.

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import lit

# Initialize Spark session

spark = SparkSession.builder.appName("schema_of_json-example").getOrCreate()

# Sample data with JSON strings

data = [

    ('{"user_id": "user_001", "name": "John Doe", "address": {"street": "123 Main St", "city": "New York"}, "is_active": true, "last_login": "2025-02-18T15:32:00"}',),

    ('{"user_id": "user_002", "name": "Alice Smith", "address": {"street": "456 Oak St", "city": "Los Angeles"}, "is_active": false, "last_login": "2025-02-18T16:45:00"}',),

    ('{"user_id": "user_003", "name": "Bob Brown", "address": {"street": "789 Pine St", "city": "San Francisco"}, "is_active": true, "last_login": "2025-02-18T17:00:00"}',)

]

# Create DataFrame with JSON string column

df = spark.createDataFrame(data, ["user_activity_json"])

# Show the DataFrame

df.show(truncate=False)
```

Output



```
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|user_activity_json                                                                                                                                                 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------+
|{"user_id": "user_001", "name": "John Doe", "address": {"street": "123 Main St", "city": "New York"}, "is_active": true, "last_login": "2025-02-18T15:32:00"}      |
|{"user_id": "user_002", "name": "Alice Smith", "address": {"street": "456 Oak St", "city": "Los Angeles"}, "is_active": false, "last_login": "2025-02-18T16:45:00"}|
|{"user_id": "user_003", "name": "Bob Brown", "address": {"street": "789 Pine St", "city": "San Francisco"}, "is_active": true, "last_login": "2025-02-18T17:00:00"}|
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

2. Use schema_of_json() to Infer the Schema

Now that we have a **DataFrame** with a **JSON column**, we can use the **schema_of_json()** function to infer the schema of the JSON string.

```
schema1=df \
.select(schema_of_json(col('user_activity_json'))).collect()[0][0]

print(schema1)
```

Output

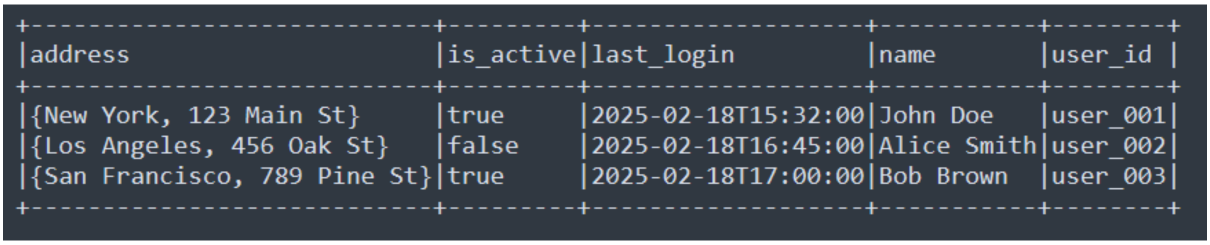
```
STRUCT<

        address: STRUCT<city: STRING, street: STRING>,

        is_active: BOOLEAN,

        last_login: STRING,

        name: STRING,

        user_id: STRING>
```

3. Apply the Inferred Schema to Parse JSON Data

Once you have inferred the schema, you can use it to **parse the JSON column** and extract the structured data. For example, you can use **from_json()** to convert the user_activity_json column into a structured format using the inferred schema.

```
from pyspark.sql.functions import from_json

# Apply the inferred schema to the JSON column

df_parsed = df.withColumn("parsed_data", from_json("user_activity_json",
schema1))

# Show the resulting DataFrame with structured data

df_parsed.select("parsed_data.*").show(truncate=False)
```

Output



```
+----------------------------+---------+-------------------+-----------+--------+
|address                     |is_active|last_login         |name       |user_id |
+----------------------------+---------+-------------------+-----------+--------+
|{New York, 123 Main St}     |true     |2025-02-18T15:32:00|John Doe   |user_001|
|{Los Angeles, 456 Oak St}   |false    |2025-02-18T16:45:00|Alice Smith|user_002|
|{San Francisco, 789 Pine St}|true     |2025-02-18T17:00:00|Bob Brown  |user_003|
+----------------------------+---------+-------------------+-----------+--------+
```

Key Points to Note:

- **schema_of_json()** is used to infer the schema of a **JSON string** column. It returns a **StructType** that describes the structure of the JSON data.
- This function is useful when working with **semi-structured JSON data**, where you may not know the schema in advance.
- Once you have the inferred schema, you can apply it using **from_json()** to convert JSON strings into structured DataFrame columns.
- You can use this method to work with large datasets where JSON data has varying or unknown structures.

Summary of Functions:

- from_json(): Parse JSON string to structured types.
- to_json(): Convert struct/map to JSON string.
- get_json_object(): Extract specific elements from JSON string.
- json_tuple(): Extract multiple fields from JSON.
- schema_of_json(): Infer schema from JSON string.

These functions make it easy to handle JSON data in PySpark, enabling you to efficiently parse, query, and transform JSON-based datasets.

By: Raushan Kumar

Please follow for more such content:

https://www.linkedin.com/in/raushan-kumar-553154297/