Master Pyspark Zero to Hero:

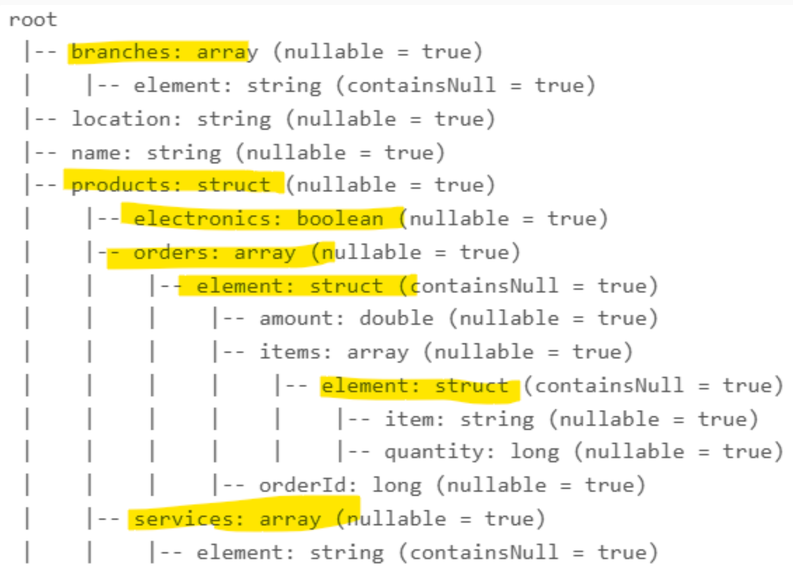
Working with Json Structure : Part 2

Exploding JSON in PySpark: Step-by-Step Explanation

To flatten and explode a nested JSON structure in PySpark, we need to work with the explode() function, which helps us expand array-type fields into individual rows. Here's how you can approach it step by step:

```
from pyspark.sql import SparkSession
import json

# Initialize Spark session
spark = SparkSession.builder.appName("FlattenJson").getOrCreate()

# Sample JSON data (Python dictionaries)
data = [
    {
        "name": "CompanyA",
        "location": "Austin",
        "branches": ["Dallas", "Houston"],
        "products": {
            "electronics": True,
            "services": ["consulting", "support"],
            "orders": [
                {"orderId": 101, "amount": 500.0, "items": [{"item": "Laptop",
"quantity": 2}, {"item": "Charger", "quantity": 3}]},
                {"orderId": 102, "amount": 300.0, "items": [{"item": "Tablet",
"quantity": 5}, {"item": "Headphones", "quantity": 1}]}
            ]
        }
    },
    {
        "name": "CompanyB",
        "location": "Dallas",
        "branches": ["Austin", "Fort Worth"],
        "products": {
            "electronics": False,
            "services": ["installation"],
            "orders": [
                {"orderId": 201, "amount": 400.0, "items": [{"item": "Printer",
"quantity": 2}]}
            ]
        }
    }
]
```

```
# Convert Python dictionaries to JSON strings
json_data = [json.dumps(record) for record in data]

# Parallelize the data and create a DataFrame
rdd = spark.sparkContext.parallelize(json_data)
df = spark.read.json(rdd)

# Display the schema and DataFrame
df.printSchema()
df.display()
```



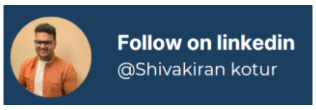
```
root
 |-- branches: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- location: string (nullable = true)
 |-- name: string (nullable = true)
 |-- products: struct (nullable = true)
 |    |-- electronics: boolean (nullable = true)
 |    |-- orders: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- amount: double (nullable = true)
 |    |    |    |-- items: array (nullable = true)
 |    |    |    |    |-- element: struct (containsNull = true)
 |    |    |    |    |    |-- item: string (nullable = true)
 |    |    |    |    |    |-- quantity: long (nullable = true)
 |    |    |    |-- orderId: long (nullable = true)
 |    |-- services: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
```

Step 1: Exploding the branches Array

The branches field in the JSON is an array of locations (e.g., "Dallas", "Houston"). The explode() function will transform this array into individual rows, with each branch becoming its own row. Other columns like name and location will be repeated for each branch.

Explanation:

- For every record in the JSON, branches is exploded into new rows, and the name and location columns are duplicated across these new rows.
- **Example:**
  - If the company has two branches, "Dallas" and "Houston", you will get two rows for that company: one for "Dallas" and another for "Houston".



Step 2: Exploding the products.services Array

The products.services field is another array, which holds service names like "consulting" or "support". We can apply the explode() function to this array as well.

Explanation:

- After exploding the branches array, we now apply explode() on the services field within the products field.
- This will create a new row for each service, duplicating the previously exploded branches and adding the service name in a new column.
- **Example:**
  - If the company offers two services, "consulting" and "support", you'll get two rows for each branch, one for each service.

Step 3: Exploding the products.orders Array

Next, the products.orders field is an array containing order details. We apply the explode() function again on this field.

Explanation:

- After exploding both branches and services, the orders array is exploded.
- Each order in the orders array (with fields like orderId and amount) gets its own row, while the name, location, branch, and service columns are repeated.
- **Example:**
  - If an order contains two items, we will now have a row for each item within that order.

Step 4: Exploding the order.items Array

Finally, the items field inside each order is also an array. The items array holds individual products like "Laptop" and "Charger", and we can apply explode() one more time to break this into separate rows.

Explanation:

- After exploding the orders, each item (e.g., "Laptop", "Charger") inside an order is now exploded into its own row.

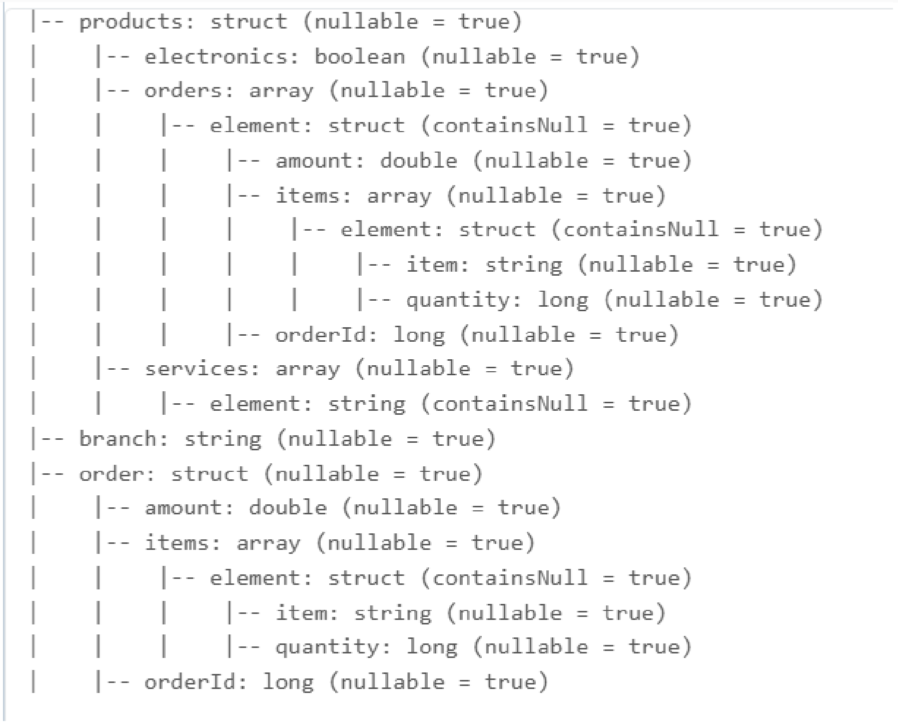
- Each item is associated with all the previous fields like name, location, branch, service, orderId, and amount, making it easier to analyze individual items in the context of the company, branch, and service.

Final Flattened Structure:

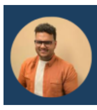
After applying explode() on all the array fields (branches, services, orders, and items), we have a fully flattened structure where each row represents:

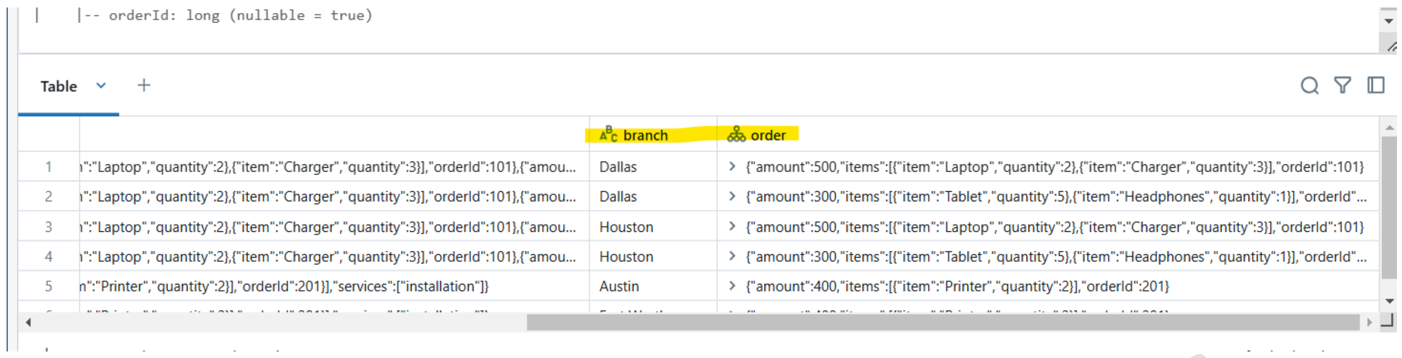
- A combination of name, location, branch, service, order details (like orderId and amount), and item details (like item_name and item_quantity).

```python
from pyspark.sql.functions import explode,col

# Explode the branches column
df_branches = df.withColumn("branch", explode(col("branches")))

# Explode the orders column inside products
df_orders = df_branches.withColumn("order",
explode(col("products.orders")))

df_orders.printSchema()
df_orders.display()
```



```
 |-- products: struct (nullable = true)
 |    |-- electronics: boolean (nullable = true)
 |    |-- orders: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- amount: double (nullable = true)
 |    |    |    |-- items: array (nullable = true)
 |    |    |    |    |-- element: struct (containsNull = true)
 |    |    |    |    |    |-- item: string (nullable = true)
 |    |    |    |    |    |-- quantity: long (nullable = true)
 |    |    |    |-- orderId: long (nullable = true)
 |    |-- services: array (nullable = true)
 |    |    |-- element: string (containsNull = true)
 |-- branch: string (nullable = true)
 |-- order: struct (nullable = true)
 |    |-- amount: double (nullable = true)
 |    |-- items: array (nullable = true)
 |    |    |-- element: struct (containsNull = true)
 |    |    |    |-- item: string (nullable = true)
 |    |    |    |-- quantity: long (nullable = true)
 |    |-- orderId: long (nullable = true)
```



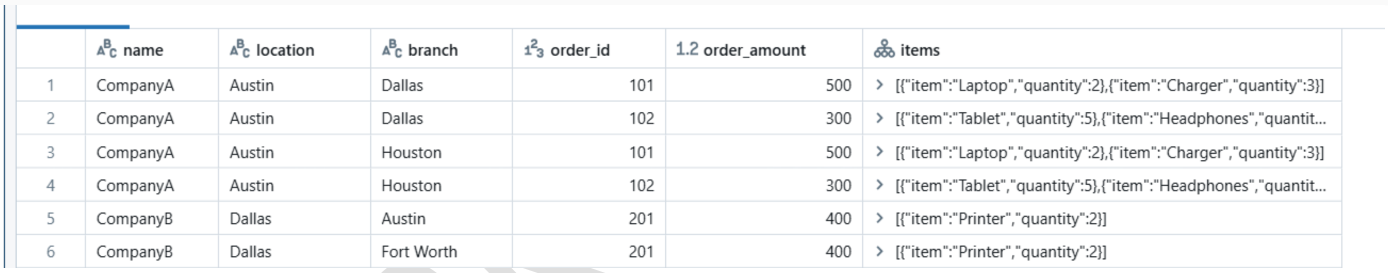


```
|    |-- orderId: long (nullable = true)
```

Table

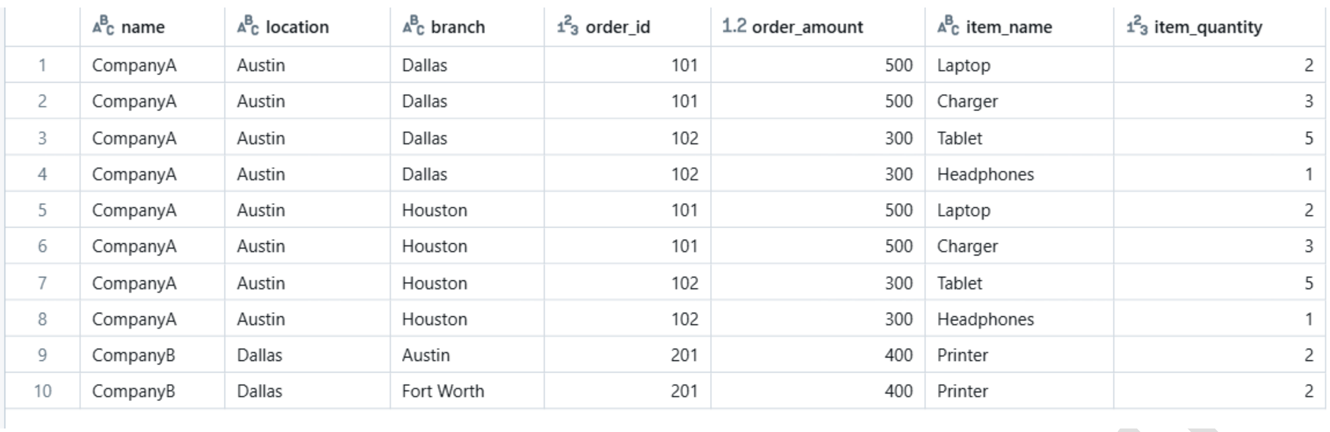
		branch	order
1	":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101},{"amou...	Dallas	> {"amount":500,"items":[{"item":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101}
2	":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101},{"amou...	Dallas	> {"amount":300,"items":[{"item":"Tablet","quantity":5},{"item":"Headphones","quantity":1}],"orderId"...
3	":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101},{"amou...	Houston	> {"amount":500,"items":[{"item":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101}
4	":"Laptop","quantity":2},{"item":"Charger","quantity":3}],"orderId":101},{"amou...	Houston	> {"amount":300,"items":[{"item":"Tablet","quantity":5},{"item":"Headphones","quantity":1}],"orderId"...
5	n":"Printer","quantity":2}],"orderId":201}],"services":["installation"]}	Austin	> {"amount":400,"items":[{"item":"Printer","quantity":2}],"orderId":201}

```python
df_flattened = df_orders.select(
    col("name"),
    col("location"),
    col("branch"),
    col("order.orderId").alias("order_id"),
    col("order.amount").alias("order_amount"),
    col("order.items").alias("items")
)


df_flattened.display()
```



	name	location	branch	order_id	order_amount	items
1	CompanyA	Austin	Dallas	101	500	> [{"item":"Laptop","quantity":2},{"item":"Charger","quantity":3}]
2	CompanyA	Austin	Dallas	102	300	> [{"item":"Tablet","quantity":5},{"item":"Headphones","quantit...
3	CompanyA	Austin	Houston	101	500	> [{"item":"Laptop","quantity":2},{"item":"Charger","quantity":3}]
4	CompanyA	Austin	Houston	102	300	> [{"item":"Tablet","quantity":5},{"item":"Headphones","quantit...
5	CompanyB	Dallas	Austin	201	400	> [{"item":"Printer","quantity":2}]
6	CompanyB	Dallas	Fort Worth	201	400	> [{"item":"Printer","quantity":2}]

```python
df_final = df_flattened.withColumn("item", explode(col("items")))

# Extract fields from the item column
df_result = df_final.select(
    col("name"),
    col("location"),
    col("branch"),
    col("order_id"),
    col("order_amount"),
    col("item.item").alias("item_name"),
    col("item.quantity").alias("item_quantity")
)

df_result.display()
```



	name	location	branch	order_id	order_amount	item_name	item_quantity
1	CompanyA	Austin	Dallas	101	500	Laptop	2
2	CompanyA	Austin	Dallas	101	500	Charger	3
3	CompanyA	Austin	Dallas	102	300	Tablet	5
4	CompanyA	Austin	Dallas	102	300	Headphones	1
5	CompanyA	Austin	Houston	101	500	Laptop	2
6	CompanyA	Austin	Houston	101	500	Charger	3
7	CompanyA	Austin	Houston	102	300	Tablet	5
8	CompanyA	Austin	Houston	102	300	Headphones	1
9	CompanyB	Dallas	Austin	201	400	Printer	2
10	CompanyB	Dallas	Fort Worth	201	400	Printer	2

Key Points:

1. **Exploding Arrays:** The explode() function is used to transform array fields into individual rows.
2. **Repetition of Non-Array Columns:** Non-array columns (like name and location) are duplicated across the new rows after each explosion.
3. **Flattening Nested Data:** By exploding nested arrays at multiple levels, you convert the original hierarchical JSON into a flat structure, making it easier to analyze and query.
4. **Multiple Explodes:** Each time you explode an array, you add new rows. You can use select() to choose which nested fields to bring to the top level, making the data more accessible.

This approach allows you to take a complex JSON structure with arrays and nested fields and convert it into a tabular format where each element is broken down into rows, making the data ready for further analysis.