How can I read a JSON file into a PySpark DataFrame?

Authored by

stats writer

June 24, 2024

To read a JSON file into a PySpark DataFrame, first import the necessary PySpark libraries and initialize a Spark session. Then, use the `.read.json()` method to load the JSON file into a DataFrame. This method automatically infers the schema and creates a DataFrame with appropriate columns and data types. Additionally, you can specify the schema manually using the `.schema()` method. Finally, use the `.show()` method to display the contents of the DataFrame. This process allows for easy and efficient access to data stored in a JSON file using PySpark.

To read JSON files into a PySpark DataFrame, users can use the `json()` method from the DataFrameReader class. This method parses JSON files and automatically infers the schema, making it convenient for handling structured and semi-structured data.

PySpark provides robust functionality for processing large-scale data, including reading data from various file formats such as JSON. JSON (JavaScript Object Notation) is a widely used format for storing and exchanging data due to its lightweight and human-readable nature.

Reading JSON files into a PySpark DataFrame enables users to perform powerful data transformations, analyses, and machine learning tasks on large datasets in a distributed computing environment. It also allows seamless integration with other PySpark operations and libraries, making it a versatile tool for big data processing pipelines.

In this article, I will explain how to utilize PySpark to efficiently read JSON files into DataFrames, how to handle null values, how to handle specific date formats, and finally, how to write DataFrame to a JSON file.

Table of contents:

Reading JSON file in PySpark

To read a JSON file into a PySpark DataFrame, initialize a SparkSession and use `spark.read.json("json_file.json")`. Replace `"json_file.json"` with the actual file path. This method automatically infers the schema and creates a DataFrame from the JSON data. Further data processing and analysis tasks can then be performed on the DataFrame.

zipcodes.json file used here can be downloaded from GitHub project.

```
# Read JSON file into dataframe
df = spark.read.json("resources/zipcodes.json")
df.printSchema()
df.show()
```

Alternatively, you can use the `format()` function along with the `load()` method to read a JSON file into a PySpark DataFrame.

```
# Read JSON file into dataframe
df = spark.read.format('org.apache.spark.sql.json')
.load("resources/zipcodes.json")
```

Reading from Multiline JSON File

To read a multiline JSON file into a PySpark DataFrame, use `spark.read.option("multiline", "true").json("path_to_json_file.json")`. This setting allows reading JSON objects spanning multiple lines. Specify the file path as `"path_to_json_file.json"`. The DataFrame is created with inferred schema, suitable for further processing.The `multiline` option is set to false by default.

Below is an example of the file content with multiline joson. You can find the file on Github.

Read this multiline JSON file into a DataFrame.

```
# Read multiline json file
multiline_df = spark.read.option("multiline","true")
.json("resources/multiline-zipcode.json")
multiline_df.show()
```

Reading from Multiple files at a time

To read multiple files at a time into a DataFrame, pass fully qualified paths by comma separated into a read.json() method. for example

```
# Read multiple files
df2 = spark.read.json(
)
df2.show()
```

Reading all files in a Folder

To read all JSON files from a directory into a PySpark DataFrame simultaneously, use `spark.read.json("directory_path")`, where `"directory_path"` points to the directory containing the JSON files. PySpark automatically processes all JSON files in the directory,

```
# Read all JSON files from a folder
df3 = spark.read.json("resources/*.json")
df3.show()
```

Reading files with a user-specified custom schema

PySpark SQL offers StructType and StructField classes, enabling users to programmatically specify the DataFrame's structure. These classes allow precise specification of column names, data types, and other attributes.

If you know the schema of the file ahead and do not want to use the default `inferSchema` option, use the schema option to specify user-defined custom column names and data types.

Use the PySpark StructType class to create a custom schema. Below, we initiate this class and use a method to add columns to it by providing the column name, data type, and nullable option.

```
# Define custom schema
schema = StructType()
```

df_with_schema = spark.read.schema(schema)
.json("resources/zipcodes.json")
df_with_schema.printSchema()
df_with_schema.show()

Reading File using PySpark SQL

PySpark SQL also provides a way to read a JSON file by creating a temporary view directly from the reading file using spark.sqlContext.sql("load JSON to temporary view")

```
spark.sql("CREATE OR REPLACE TEMPORARY VIEW zipcode USING json OPTIONS" +
" (path 'resources/zipcodes.json')")
spark.sql("select * from zipcode").show()
```

JSON File Reading Options

nullValues

The `nullValues` option in PySpark is used to specify custom strings that should be treated as null values during the data ingestion process. For example, if you want to consider a field with a value "N/A" as null on DataFrame.

dateFormat

The `dateFormat` option is used to specify the format of date or timestamp columns in the input data. This option allows PySpark to correctly parse date or timestamp strings into their corresponding data types. Supports all java.text.SimpleDateFormat formats.

Note: These are just a few options, for the complete list, refer to Spark's official documentation

Applying DataFrame transformations

After reading a JSON file into a DataFrame in PySpark, we typically apply the transformations that allow us to manipulate, clean, or preprocess the data according to analysis or processing requirements. Some common reasons for applying transformations include:

Data Cleaning: Transformations can be used to clean the data by handling missing values, filtering out irrelevant rows, or correcting inconsistencies in the data.**Data Enrichment**: You can enrich the data by adding new columns, aggregating information, or joining with other datasets to provide additional context or insights.**Data Formatting**: Transformations enable you to format the data in a desired way, such as converting data types, renaming columns, or applying custom formatting to values.**Data Aggregation**: Aggregating the data allows you to summarize information, calculate statistics, or group data based on specific criteria.**Feature Engineering**: Transformations are often used in feature engineering to create new features or modify existing ones to improve model performance in machine learning tasks.**Data Exploration**: Transformations facilitate exploratory data analysis by reshaping the data, extracting subsets of interest, or visualizing patterns to gain insights into the dataset.

Refer to PySpark Transformations for examples

Write PySpark DataFrame to JSON file

To write a DataFrame to a JSON file in PySpark, use the `write.json()` method and specify the path where the JSON file should be saved. Optionally, you can also specify additional options such

as the mode for handling existing files and compression type. Note that `write` is an object of `DataFrameWriter` class.

```
# Write dataframe to json file
df2.write.json("/tmp/spark_output/zipcodes.json")
```

This command writes the DataFrame `df2` to the specified output path as JSON files. Each partition of the DataFrame is written as a separate JSON file. The data is serialized in JSON format, preserving the DataFrame's schema.

Options while writing JSON files

When writing a DataFrame to JSON files in PySpark, you can specify options to specify how you wante to write the files. Some commonly used options include:

`path`: Specifies the path where the JSON files will be saved.`mode`: Specifies the behavior when writing to an existing directory.`compression`: Specifies the compression codec to use when writing the JSON files (e.g., "gzip", "snappy").`dateFormat`: Specifies the format for date and timestamp columns.`timestampFormat`: Specifies the format for timestamp columns.`lineSep`: Specifies the character sequence to use as a line separator between JSON objects.`encoding`: Specifies the character encoding to use when writing the JSON files.

```
df2.write
.option("compression", "gzip")
.option("dateFormat", "yyyy-MM-dd")
.json("output_path")
```

PySpark Saving modes

In PySpark, when saving DataFrames to external storage such as file systems or databases, different saving modes can be specified to control the behavior in case the target location already exists. The saving modes include:

Append: Appends the data to the existing data in the target location. If the target location does not exist, it creates a new one.**Overwrite**: Overwrites the data in the target location if it already exists. If the target location does not exist, it creates a new one.**Ignore**: Ignores the operation and does nothing if the target location already exists. If the target location does not exist, it creates a new one.**Error or ErrorIfExists**: Throws an error and fails the operation if the target location already exists. This is the default behavior if no saving mode is specified.

These saving modes provide flexibility and control over how data is saved and handled in different scenarios, ensuring data integrity and consistency in data processing workflows.

```
# Write with savemode example
df2.write.mode('Overwrite').json("/tmp/spark_output/zipcodes.json")
```

Source code for reference

This example is also available at GitHub PySpark Example Project for reference.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType,
IntegerType,BooleanType,DoubleType
spark = SparkSession.builder
.master("local")
.appName(arabpsychology.com)
.getOrCreate()

# Read JSON file into dataframe
df = spark.read.json("resources/zipcodes.json")
df.printSchema()
df.show()

# Read multiline json file
multiline_df = spark.read.option("multiline","true")
.json("resources/multiline-zipcode.json")
multiline_df.show()

#Read multiple files
df2 = spark.read.json(
)
df2.show()

#Read All JSON files from a directory
df3 = spark.read.json("resources/*.json")
df3.show()

# Define custom schema
schema = StructType()
```

```
df_with_schema = spark.read.schema(schema)
.json("resources/zipcodes.json")
df_with_schema.printSchema()
df_with_schema.show()

# Create a table from Parquet File
spark.sql("CREATE OR REPLACE TEMPORARY VIEW zipcode3 USING json OPTIONS" +
" (path 'resources/zipcodes.json')")
spark.sql("select * from zipcode3").show()

# PySpark write Parquet File
df2.write.mode('Overwrite').json("/tmp/spark_output/zipcodes.json")
```

Frequently Asked Questions on PySpark Read JSON

Can we read multiple JSON files into a single DataFrame?

We can read multiple JSON files into a single DataFrame by providing a directory path containing the JSON files. PySpark will automatically combine them into one DataFrame.

For example:

```
df = spark.read.json("path/to/json/files/")
```

How can I specify a schema while reading JSON data?

If you know the schema of the file ahead and do not want to use the default `inferSchema` option, use the schema option to specify user-defined custom column names and data types.

How to handle the schema of JSON data that has nested structures?

PySpark can handle nested structures in JSON data. The `spark.read.json()` method automatically infers the schema, including nested structures. You can access nested fields using dot notation in DataFrame queries.

What if my JSON data is not in a file but stored in a variable?

If your JSON data is stored in a variable, you can use the `spark.read.json()` method with the `jsonRDD` method. For Example:

```
json_object = '{"name": Cinthia", "age": 20}'
df = spark.read.json(spark.sparkContext.parallelize())
```

Conclusion

In conclusion, PySpark provides powerful capabilities for reading and writing JSON files, facilitating seamless integration with various data sources and formats. By leveraging JSON API and numerous options, users can efficiently ingest JSON data into DataFrames. PySpark's flexible JSON writing functionality empowers users to export processed data back to JSON files, preserving schema.

Throughout this tutorial, you've gained insights into reading JSON files with both single-line and multiline records into PySpark DataFrame. Additionally, you've learned techniques for reading single and multiple files simultaneously, as well as methods for writing DataFrame data back into JSON files.

Related Articles

References:

Happy Learning !!