BA-BEAD Big Data Engineering for Analytics

Workshop Series

Data Format

Table of Contents

List of python scripts and datasets

Scripts	Datasets
ReadCSV.py	people.csv
WriteCSV.py	NA
ReadJSON.py	people.json
Parquet_Example_python.py	resale-flat-prices-based-on-registration-date-from-mar-2012-to-dec-2014.csv
Arrow_Example.py	NA

Introduction

In this workshop, we will use PyCharm IDE to read and write different data format files. PySpark provides interface used to load **`DataFrame`** from external storage systems. We will learn how to read different data format files into **`DataFrame`** and write **`DataFrame`** back to different data format files using PySpark examples. Lastly, we will learn how to transfer data between JVM and Python processes using Apache Arrow efficiently.

Create New Project in PyCharm

1. Click “Create New Project” in the PyCharm welcome screen.

2. Give a meaningful project name, such as “DataFormat”:

3. Now that we have created a Python project named as “DataFormat”, next step we need to create a Python program file to write and run Python programs. To create a file, right click on File -> New -> Python file. Give the file name such as “ReadCSV”.

Read CSV file

PySpark provides **`DataFrameReader`** to load a **`DataFrame`** from external storage systems (e.g. file systems, key-value stores, etc). Use **`SparkSession.read`** to access this. You can use **`format(source)`** to specify the input data source format.

Using **`csv("path")`** or **`format("csv").load("path")`** of **`DataFrameReader`**, you can read a CSV file into a PySpark **`DataFrame`**, These methods take a file path to read from as an argument. When you use **`format("csv")`** method, you can also specify the data sources by their fully qualified name, but for built-in sources, you can simply use their short names (csv,json, parquet, jdbc, text e.t.c).

In this example, it shows how to read a single CSV file “people.csv” into **`DataFrame`** as well as how to use your own defined schema when read file into **`DataFrame`**.

ReadCSV.py:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Read CSV files').getOrCreate()
from pyspark.sql.types import StructType, StructField,
StringType,IntegerType

# Read CSV file people.csv
df = spark.read.format('csv') \
                .option("inferSchema","true") \
                .option("header","true") \
                .option("sep",";") \
                .load("people.csv")

# Show result
df.show()
# Print schema
df.printSchema()

# Define your own schema
schema = StructType([ \
    StructField("name",StringType(),True), \
    StructField("age",IntegerType(),True), \
    StructField("job",StringType(),True)])
peopleDF = spark.read.format('csv') \
                    .schema(schema) \
                    .option("sep",";") \
                    .option("header","true") \
                    .load("people.csv")

#peopleDF = spark.read.load("people.csv", format = "csv", header =
"true",sep=";",schema=schema)
peopleDF.show()
peopleDF.printSchema()
```

The output is:

```
+-----+---+---------+
| name|age|    job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+
```

```
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)

+-----+---+---------+
| name|age|      job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+

root
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)
```

Write CSV file (Optional)

PySpark provides interface **`DataFrameWriter`** to write a **`DataFrame`** to external storage systems. Use **`DataFrame.write`** to access this. It can write into different formats such as csv, json, parquet, etc.

While writing a CSV file you can use several options. For example, header to output the **`DataFrame`** column names as header record and delimiter to specify the delimiter on the CSV output file.

In the following example, a **`DataFrame`** is read from a CSV file. The **`DataFrame`** is then saved to the current local file path. To save file to local path, specify 'file://'. The file can also save into HDFS. By default, the path is HDFS path. There are several options used:

1. **`header`**: to specify whether include header in the file.
2. **`sep`**: to specify the delimiter
3. **`mode`** is used to specify the behavior of the save operation when data already exists.
    - *append*: Append contents of this DataFrame to existing data.
    - *overwrite*: Overwrite existing data.
    - *error* or *errorifexists*: Throw an exception if data already exists.*ignore*: Silently ignore this operation if data already exists.
    - *ignore:* Silently ignore this operation if data already exists.

WriteCSV.py:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Write CSV files').getOrCreate()


# Write csv file
df = spark.read.format('csv') \
                .option("inferSchema","true") \
                .option("header","true") \
                .option("sep",";") \
                .load("people.csv")

# Show result
df.show()
# Print schema
df.printSchema()

df.write.format("csv") \
                .option("header","true") \
                .option("sep",";")  \
                .mode("overwrite") \
                .save("people2")
```

Please note that the above codes will create a folder to save the file. If there is a large file with multiple partitions, it will create a folder with multiple files, because each partition is saved individually.

Read JSON file

You can read JSON file by specifying `format('json').` In this example, it shows how to read a json format file.

ReadJSON.py:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('read JSON files').getOrCreate()


df = spark.read.format('json') \
                .option("inferSchema","true") \
                .option("header","true") \
                .option("sep",";") \
                .load("people.json")
df.show()
df.printSchema

df2 = spark.read.json("people.json")
df2.show()
```

Read and Write Parquet file

In this example, we read a csv format file and convert it to a parquet file, and inspect the parquet metadata.

Parquet_Example_python.py

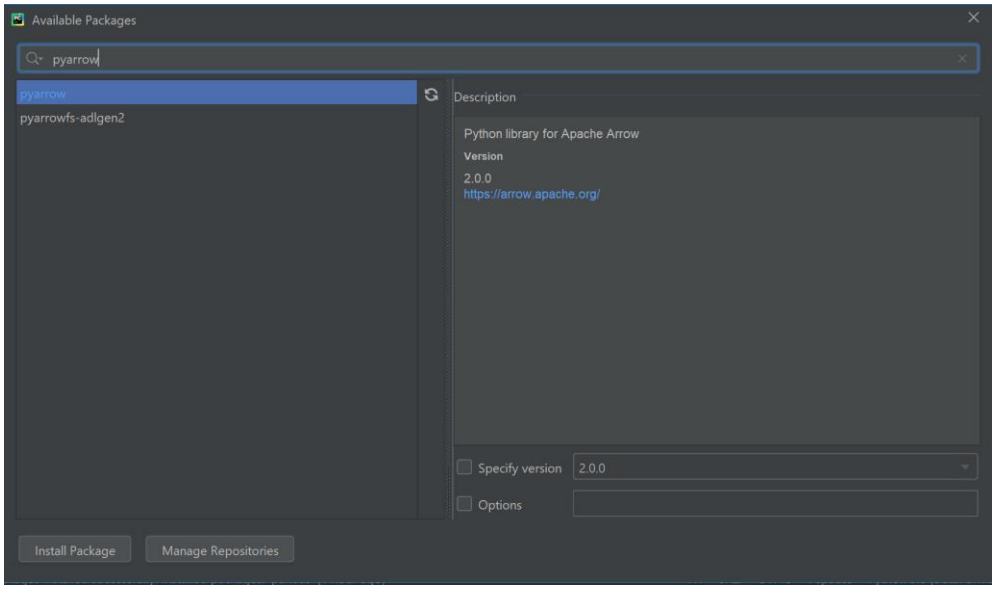
```
import pyarrow.csv as pv

import pyarrow.parquet as pq

# read hdb resale price

hdb_table = pv.read_csv("resale-flat-prices-based-on-registration-date-from-mar-2012-to-dec-2014.csv")

# convert the CSV file to a Parquet file

pq.write_table(hdb_table,'resale-flat-prices-based-on-registration-date-from-mar-2012-to-dec-2014.parquet')

hdb_parquet = pq.ParquetFile('resale-flat-prices-based-on-registration-date-from-mar-2012-to-dec-2014.parquet')

# inspect the parquet metadata

print(hdb_parquet.metadata)

# inspect the parquet row group metadata

print(hdb_parquet.metadata.row_group(0))

# inspect the column chunk metadata

print(hdb_parquet.metadata.row_group(0).column(9).statistics)
```

Arrow Example

Apache Arrow is an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes. This currently is most beneficial to Python users that work with Pandas/NumPy data.

In order to use PyArrow, we need to install PyArrow package.

Go to File –> Setting –> Python Interpret - > search for PyArrow, and install the package.



Arrow is available as an optimization when converting a Spark **`DataFrame`** to a Pandas **`DataFrame`** using the call **`toPandas`**() and when creating a Spark **`DataFrame`** from a Pandas **`DataFrame`** with **`createDataFrame(pandas_df)`**. To use Arrow when executing these calls, we need to first set the Spark configuration **`spark.sql.execution.arrow.pyspark.enabled`** to **`true`**. This is disabled by default.

In this example, we create a Spark **`DataFrame`** from a Pandas **`DataFrame`** and convert the Spark **`DataFrame`** back to a Pandas **`DataFrame`** using Arrow. Arrow_Example.py:

```
import numpy as np
import pandas as pd
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Arrow example').getOrCreate()
# Enable Arrow-based columnar data transfers
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# Generate a Pandas DataFrame
pdf = pd.DataFrame(np.random.rand(100, 3))

# Create a Spark DataFrame from a Pandas DataFrame using Arrow
df = spark.createDataFrame(pdf)

# Convert the Spark DataFrame back to a Pandas DataFrame using Arrow
result_pdf = df.select("*").toPandas()

print(result_pdf)
```

The output is:

```
      0     1     2
```

```
0   0.726908  0.271894  0.728025
1   0.889413  0.241705  0.944082
2   0.393902  0.882159  0.252503
3   0.235247  0.634970  0.558968
4   0.408405  0.698789  0.695485
..       ...       ...       ...
95  0.316589  0.609944  0.659414
96  0.972518  0.410238  0.966415
97  0.959871  0.092835  0.062921
98  0.236058  0.690564  0.205223
99  0.082190  0.794287  0.606402

[100 rows x 3 columns]
```

~~End~~