Basic DF Operations

June 21, 2020

1 Basic Dataframes operations

[1]:
```
import pyspark
from pyspark.sql import SparkSession

sc = pyspark.SparkContext()
spark = SparkSession(sc)
```

1.1 Schemas

Schema can be inferred automatedly. Spark also can infer schema from semi-structured data formats, e.g. JSON

[2]:
```
spark.read.format("json").load('../data/flights.json').schema
```

[2]:
```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField(ORIGI
N_COUNTRY_NAME,StringType,true),StructField(count,LongType,true)))
```

Alternatively, we also can define schema explicitly. Automated schema discovery is not always robust, sometimes struggling to detect the right type of each columns, e.g. precision issues (int detected instead of long). Moreover, inferring schema sometimes can be a bit slolw.

Parameter of StructField:

name – string, name of the field.

dataType – DataType of the field.

nullable – boolean, whether the field can be null (None) or not.

metadata – a dict from string to simple type that can be toInternald to JSON automatically

[3]:
```
from pyspark.sql.types import StructField, StructType, StringType, LongType
manualSchema = StructType([
    # StructField (name, dataType, nullable, metadata)
    StructField("DEST_COUNTRY_NAME", StringType(), True),
    StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
    StructField("count", LongType(), False)
])
```

```
flightsDF = spark.read.format("json").schema(manualSchema).load('../data/
 ↪flights.json')
flightsDF.schema
```

[3]:
```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),StructField(ORIGI
N_COUNTRY_NAME,StringType,true),StructField(count,LongType,true)))
```

1.2 Creating dataframes

There are several ways for creating dataframes. The most straightforward way is to create a dataframe from the content of file(s). However, it is also possible to create dataframes programatically.

[4]:
```
from pyspark.sql import Row
mySchema = StructType([
    StructField("colA", StringType(), True),
    StructField("colB", LongType(), True)
])

# DFs basically are just collections of Rows. We can create a Row object in the␣
 ↪following way:
myRow = Row("Hello", 12)
myDF = spark.createDataFrame([myRow], mySchema)
myDF.show()
```

```
+-----+----+
| colA|colB|
+-----+----+
|Hello|  12|
+-----+----+
```

We can also create a Dataframe from RDD, if our RDD elements are also Row objects

[5]:
```
collection = [Row("Hello!", 12), ("Hallo!", 23)]
rdd = spark.sparkContext.parallelize(collection)
df = spark.createDataFrame(rdd, mySchema)
df.show()
```

```
+------+----+
|  colA|colB|
+------+----+
|Hello!|  12|
|Hallo!|  23|
+------+----+
```

1.3 Projection (selection of columns)

Projection allows you to select specific columns only from the DF. This is very similar to SQL SELECT

[6]:
```
flightsDF.select("DEST_COUNTRY_NAME").show(2)
```

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
only showing top 2 rows
```

[7]:
```
flightsDF.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
```

```
+-----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+-----------------+-------------------+
only showing top 2 rows
```

[8]:
```
from pyspark.sql.functions import expr, col, column
flightsDF.select(
    column("DEST_COUNTRY_NAME"),
    # flexible referencing for columns
    expr("DEST_COUNTRY_NAME as destination"),
    # with expr() we also can apply string manipulation
    expr("upper(DEST_COUNTRY_NAME) as u_destination")
).show(2)
```

```
+-----------------+-------------+-------------+
|DEST_COUNTRY_NAME|  destination|u_destination|
+-----------------+-------------+-------------+
|    United States|United States|UNITED STATES|
|    United States|United States|UNITED STATES|
+-----------------+-------------+-------------+
only showing top 2 rows
```

Use the **selectExpr()** method when you want to specify aggregations over the entire DataFrame, e.g. calculate the avg value of a specific columns or count the number of distinct values

[9]:
```
flightsDF.selectExpr("avg(count) as avg_count",␣
 ↪"count(distinct(DEST_COUNTRY_NAME)) as num_of_dest_countries").show()
```

```
+-----------+---------------------+
|  avg_count|num_of_dest_countries|
+-----------+---------------------+
|1770.765625|                  132|
+-----------+---------------------+
```

1.4 Selection / Filtering

[10]:
```
# The following statements aree the same:
flightsDF.filter(col("count") < 2).show(10)
flightsDF.where("count < 2").show(10)
```

```
+-------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-------------------+-------------------+-----+
|       United States|            Croatia|    1|
|       United States|          Singapore|    1|
|             Moldova|      United States|    1|
|               Malta|      United States|    1|
|       United States|          Gibraltar|    1|
|Saint Vincent and…|      United States|    1|
|            Suriname|      United States|    1|
|       United States|             Cyprus|    1|
|        Burkina Faso|      United States|    1|
|            Djibouti|      United States|    1|
+-------------------+-------------------+-----+
only showing top 10 rows

+-------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-------------------+-------------------+-----+
|       United States|            Croatia|    1|
|       United States|          Singapore|    1|
|             Moldova|      United States|    1|
|               Malta|      United States|    1|
|       United States|          Gibraltar|    1|
|Saint Vincent and…|      United States|    1|
|            Suriname|      United States|    1|
|       United States|             Cyprus|    1|
|        Burkina Faso|      United States|    1|
|            Djibouti|      United States|    1|
+-------------------+-------------------+-----+
```

```
only showing top 10 rows
```

1.5 Sorting

Keep in mind, sorting is a wide operation, requires data exchange among Spark worker nodes.

[11]:
```
from pyspark.sql.functions import desc, asc
flightsDF.sort("count").show(5)
flightsDF.sort(column("count").desc()).show(5)
```

```
+--------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+--------------------+-------------------+-----+
|               Malta|      United States|    1|
|Saint Vincent and…|      United States|    1|
|       United States|            Croatia|    1|
|       United States|          Gibraltar|    1|
|       United States|          Singapore|    1|
+--------------------+-------------------+-----+
only showing top 5 rows

+-----------------+-------------------+------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+-----------------+-------------------+------+
|    United States|      United States|370002|
|    United States|             Canada|  8483|
|           Canada|      United States|  8399|
|    United States|             Mexico|  7187|
|           Mexico|      United States|  7140|
+-----------------+-------------------+------+
only showing top 5 rows
```

[12]:
```
flightsDF.orderBy(expr("count desc")).show(5)
```

```
+--------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+--------------------+-------------------+-----+
|               Malta|      United States|    1|
|Saint Vincent and…|      United States|    1|
|       United States|            Croatia|    1|
|       United States|          Gibraltar|    1|
|       United States|          Singapore|    1|
+--------------------+-------------------+-----+
only showing top 5 rows
```

1.6 Working with Nulls

Nulls are a challenging part of all programming, and Spark is no exception. In our opinion, being explicit is always better than being implicit when handling null values. For instance, in this part of the book, we saw how we can define columns as having null types. However, this comes with a catch. When we declare a column as not having a null time, that is not actually enforced. To reiterate, when you define a schema in which all columns are declared to not have null values, Spark will not enforce that and will happily let null values into that column. The nullable signal is simply to help Spark SQL optimize for handling that column. If you have null values in columns that should not have null values, you can get an incorrect result or see strange exceptions that can be difficult to debug.

[13]:
```
myRow1 = Row("Hello", None)
myRow2 = Row(None, 12)
myDF = spark.createDataFrame([myRow1, myRow2], mySchema)
```

coalesce(): returns the first non-null value from a column (this is a different coalesce() from what we used in the previous section for combining partitions!)

[14]:
```
from pyspark.sql.functions import coalesce
myDF.select(column("colA"), column("colB"), coalesce(column("colA"),␣
 ↪column("colB"))).show()
```

```
+-----+----+--------------------+
| colA|colB|coalesce(colA, colB)|
+-----+----+--------------------+
|Hello|null|               Hello|
| null|  12|                  12|
+-----+----+--------------------+
```

[15]:
```
# Just to show the same in SQL
myDF.createOrReplaceTempView("df")
spark.sql("""

SELECT colA, colB, coalesce(colA, colB)
FROM df

""").show()
```

```
+-----+----+-------------------------------------+
| colA|colB|coalesce(colA, CAST(colB AS STRING))|
+-----+----+-------------------------------------+
|Hello|null|                                Hello|
| null|  12|                                   12|
+-----+----+-------------------------------------+
```

Some other useful functions:

ifnull(val1, val2): allows you to use val2 is val1 == null

nullif(val1, val2): if val1 == val2 then returns null or else returns val2

nvl2(val1, val2, val3): if val1 == null: return val2 else return val3

[16]:
```
spark.sql("""

SELECT
    ifnull(null, 'return_value'),
    nullif('value', 'value'),
    nvl2('not_a_null_value', 'return_value', 'else_value')
FROM df LIMIT 1

""").show()
```

```
+-----------------------------+------------------------+-------------------------
-----------------------------+
|ifnull(NULL, 'return_value')|nullif('value', 'value')|nvl2('not_a_null_value',
'return_value', 'else_value')|
+-----------------------------+------------------------+-------------------------
-----------------------------+
|                 return_value|                    null|
return_value|
+-----------------------------+------------------------+-------------------------
-----------------------------+
```

Use the **drop()** operation to remove any rows which contain null value:

[17]:
```
myDF.show()
myDF.na.drop().show()
```

```
+-----+----+
| colA|colB|
+-----+----+
|Hello|null|
| null|  12|
+-----+----+

+----+----+
|colA|colB|
+----+----+
+----+----+
```

[18]:
```
# or define specific columns to drop
myDF.na.drop("all", subset=['ColB']).show()
```

```
+----+----+
```

```
|colA|colB|
+----+----+
|null|  12|
+----+----+
```

[19]:
```
# all vs any
myDF.na.drop("all").show()
myDF.na.drop("any").show()
```

```
+-----+----+
| colA|colB|
+-----+----+
|Hello|null|
| null|  12|
+-----+----+

+----+----+
|colA|colB|
+----+----+
+----+----+
```