Data manipulation with DataFrames

INTRODUCTION TO PYSPARK

Ben Schmidt
Data Engineer

datacamp

Handling missing data

- Use `.na.drop()` to remove rows with null values

```
# Drop rows with any nulls
df_cleaned = df.na.drop()

# Filter out nulls
df_cleaned = df.where(col("columnName").isNotNull())
```

- Use `.na.fill({"column": value)` to replace nulls with a specific value

```
# Fill nulls in the age column with the value 0
df_filled = df.na.fill({"age": 0})
```

Column operations

- Use `.withColumn()` to add a new column based on calculations or existing columns

```
# Create a new column 'age_plus_5'
df = df.withColumn("age_plus_5", df["age"] + 5)
```

- Use `withColumnRenamed()` to rename columns

```
# Rename the 'age' column to 'years'
df = df.withColumnRenamed("age", "years")
```

- Use `drop()` to remove unnecessary columns

```
# Drop the 'department' column
df = df.drop("department")
```

Row operations

- Use `.filter()` to select rows based on specific conditions

```
# Filter rows where salary is greater than 50000
filtered_df = df.filter(df["salary"] > 50000)
```

- Use `.groupBy()` and aggregate functions (e.g., `.sum()` , `.avg()`) to summarize data

```
# Group by department and calculate the average salary
grouped_df = df.groupBy("department").avg("salary")
```

Row Operations Outomes

- Filtering

```
+------+---+-------------------+
|salary|age|      occupation |
+------+---+-------------------+
| 60000| 45|Exec-managerial  |
| 70000| 35|Prof-specialty   |
+------+---+-------------------+
```

- GroupBy ` +----------+-----------+ |department|avg(salary)| +----------+-----------+ | HR| 80000.0| | IT| 70000.0| +----------+-----------+

`

CheatSheet

```
# Drop rows with any nulls
df_cleaned = df.na.drop()
#Drop nulls on a column
df_cleaned = df.where(col("columnName").isNotNull())
# Fill nulls in the age column with the value 0
df_filled = df.na.fill({"age": 0})
```

- Use `.withColumn()` to add a new column based on calculations or existing columns. Syntax: .withColumn("new_col_name", "original transformation")

```
# Create a new column 'age_plus_5'
df = df.withColumn("age_plus_5", df["age"] + 5)
```

- Use `withColumnRenamed()` to rename columns Syntax: `withColumnRenamed(` old column name `,` new column name`

```
# Rename the 'age' column to 'years'
df = df.withColumnRenamed("age", "years")
```

- Use `drop()` to remove unnecessary columns Syntax: .drop(`column name`)

Let's practice!

INTRODUCTION TO PYSPARK

Advanced DataFrame operations

INTRODUCTION TO PYSPARK

Ben Schmidt
Data Engineer

datacamp

Joins in PySpark

- Combine rows from two or more DataFrames based on common columns
- Types of joins: inner, left, right, and outer, like SQL
- Syntax: `DataFrame1.join(DataFrame2, on="column", how="join_type")`

```
# Joining on id column using an inner join
df_joined = df1.join(df2, on="id", how="inner")

# Joining on columns with different names
df_joined = df1.join(df2,
               df1.Id == df2.Name,
               "inner")
```

Union operation

- Combines rows from two DataFrames with the same schema
- Syntax: `DataFrame1.union(DataFrame2)`

```
# Union of two DataFrames with identical schemas
df_union = df1.union(df2)
```

Working with Arrays and Maps

Arrays: Useful for storing lists within columns, syntax: `ArrayType(StringType(),False)``

```
from pyspark.sql.functions import array, struct, lit

# Create an array column
df = df.withColumn("scores", array(lit(85), lit(90), lit(78)))
```

Maps: Key-value pairs, helpful for dictionary-like data, `MapType(StringType(),StringType())`

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

schema = StructType([
    StructField('name', StringType(), True),
    StructField('properties', MapType(StringType(), StringType()), True)
])
```

Working with Structs

- Structs: Create nested structures within rows Syntax: `StructType(Structfield, Datatype())`

```
# Create a struct column
df = df.withColumn("name_struct", struct("first_name", "last_name"))
# Create a struct column
df = df.withColumn("name_struct", struct("first_name", "last_name"))
```

Let's practice!

INTRODUCTION TO PYSPARK

U define it? U use it!

INTRODUCTION TO PYSPARK

Benjamin Schmidt
Data Engineer

datacamp

UDFs for repeatable tasks

UDF (User-Defined Function): custom function to work with data using PySpark dataframes

Advantages of UDFs:

- Reuse and repeat common tasks
- Registered directly with Spark and can be shared
- PySpark DataFrames (for smaller datasets)
- pandas UDFs (for larger datasets)

Defining and registering a UDF

All PySpark UDFs need to be registered via the `udf()` function.

```
# Define the function
def to_uppercase(s):
    return s.upper() if s else None

# Register the function
to_uppercase_udf = udf(to_uppercase, StringType())

# Apply the UDF to the DataFrame
df = df.withColumn("name_upper", to_uppercase_udf(df["name"]))

# See the results
df.show()
```

Remember: UDFs allow you to apply custom Python logic on PySpark DataFrames

pandas UDF

- Eliminates costly conversions of code and data
- Does not need to be registered to the SparkSession
- Uses pandas capabilities on extremely large datasets

```
from pyspark.sql.functions import pandas_udf

@pandas_udf("float")
def fahrenheit_to_celsius_pandas(temp_f):
    return (temp_f - 32) * 5.0/9.0
```

PySpark UDFS vs. pandas UDFs

PySpark UDF

- Best for relatively small datasets
- Simple transformations like data cleaning
- Changes occur at the columnar level, not the row level
- Must be registered to a Spark Session with `udf()`

pandas UDF

- Relatively large datasets
- Complex operations beyond simple data cleaning
- Specific row level changes over column level
- Can be called outside the Spark Session

Let's practice!

INTRODUCTION TO PYSPARK

datacamp