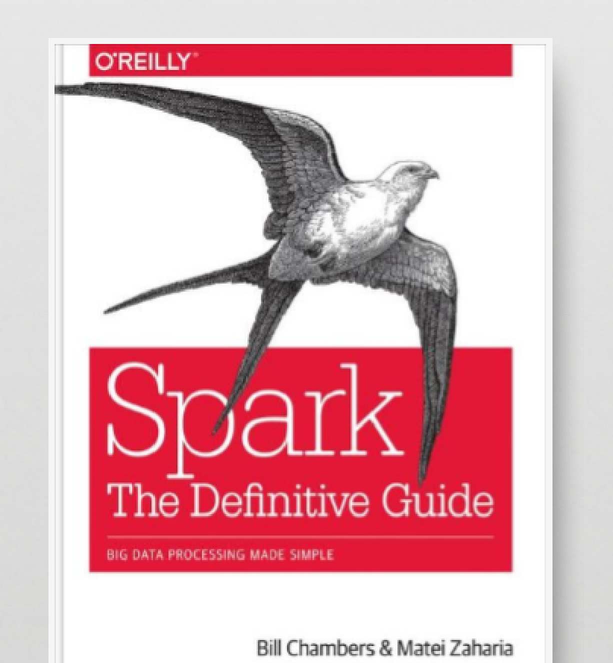
Spark the Definitive Guide 2nd Edition

Chapter 05

Basic Structured Operations

Basic Structured Operations

Text Book


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia

Objectives and Outcomes

- Introduce the tools we will use to manipulate DataFrames
- Focus on fundamental DataFrame operations
  - Understand what an expression is
  - Understand the difference between Select and SelectExpr
  - Understand how to add columns and rows to a DataFrame
  - Understand how to take random samples from DataFrames

Review

So far:

- We were introduced to Spark's Structured APIs, Datasets, DataFrames, and SQL Views
- We learned how Spark transforms a logical plan into a physical execution plan on a cluster
- Learned how DataFrames consist of a series of records
- Learned how DataFrames are of type `Row` and have a number of columns
- Learned that schemas define the name and type of data in each column
- Learned that `Partitioning` of the DataFrame defines the layout of the DataFrames physical distribution on the cluster

Create a DataFrame

```
df = spark.read.format("json").load("Spark-The-Definitive-(

# Let's tale a look at the defined schema
df.printSchema()
```

- JSON
  - is a lightweight, text-based data interchange format.

Schemas

- Schemas tie everything together
- Schema defines the column names and column types of a DataFrame
  - Schema can be applied on read or inferred or declared
- For Ad-hoc data usually *schema-on-read* is good enough
  - Though it can be a bit slow when dealing with text-based file formats like:
    - CSV
    - JSON
- Schema-on-read can lead to precision problems
  - If a column is really of type `LONG` but the numbers are smaller and interpreted as type `INT`
- Spark can be used for **ETL**:
  - Extraction
  - Transform
  - Load In these cases it is best to provide the schema to ensure type matches

JSON Object

```
spark.read.format("json").load("Spark-The-Definitive-Guide/

# This datatype is returned from the previous command
# StructType(List(StructField
# (DEST_COUNTRY_NAME,StringType,true),
# StructField
# (ORIGIN_COUNTRY_NAME,StringType,true),
# StructField(count,LongType,true)))
```

- A schema is a `StructType` made up of a number of fields
  - `StructFields` have a name, type, and b a Boolean flag indicating if they take `nulls`
  - If types in the data at run-time do not match the schema, Spark will thrown and error

Declare a Schema

```
from pyspark.sql.types import StructField, StructType, Str

myManualSchema = StructType([StructField("DEST_COUNTRY_NAM
StructField("ORIGIN_COUTNRY_NAME",StringType(),True),Struct

df = spark.read.format("json").schema(myManualSchema)
.load("Spark-The-Definitive-Guide/data/flight-data/json/20
```

Columns and Expressions

- Columns can be selected, manipulated, and removed from DataFrames
  - These operations are referred to as *expressions*
  - Must use Spark to manipulate Rows (logical collection of Rows is a column)
  - Must be in the context of a DataFrame
  - To work on columns use the *col* or *column* functions
  - We will stick to using the *col* function
  - Columns are not resolved until compared to the *catalog* at run-time
  - Column and table resolution happen in the *analyzer* phase

```
from pyspark.sql.functions import col, column

col("someColumnName")
column("someColumnName")
```

Column Reference

- If you need to explicitly reference a column you can
- Think of it as a namespace way to reference columns in different DataFrames that have the same name
  - `df.col("count")`

Columns as Expressions

- What is an *expression*?
  - A set of *transformations* on one or more values in a *record* in a DataFrame
- You can use a `col()` and perform a transformation on a column
- You can use an `expr()` to parse transformations and column references
  - These references can subsequently be passed into further transformations
  - `expr("someCol - 5")` and `col("someCol") - 5` and `expr("someCol") - 5` all evaluate the same
  - Spark compiles these to the same logic tree
- Columns are just expressions
- Columns and transformations of those columns compile to the same logical plan
  - `(((col("someCol") + 5 ) * 200 ) - 6 ) < col("otherCol")`

Directed Acyclic Graph

- This is also represented by in Python (64):
  - ```
    from pyspark.sql.functions import expr expr("(((someCol
    ```
  - Previous expression is actually valid SQL code
- This means you can write your expressions as DataFrame code or as SQL expressions and get the same performance characteristics
    ```

Accessing a DataFrames Columns

- How can you see a DataFrame's columns?
  `spark.read.format("json").load("The-Definitive-Guide-To`

Records and Rows 65

- Review: Each row in a DataFrame is a single record
  - Represented as an object of type `Row`
- How to read the first row of a DataFrame:
  - `df.first()`
- Only DataFrames have schemas, Rows do not have a schema
- To create a *Row* you must append values in the correct "schema"
  - `from pyspark.sql import Row`
  - `myRow = Row("Hello", None, 1, False)`
- To access Rows, Python and R will autodetect the datatype
  - myRow2
  - myRow0
- Scala and Java will require casting or coercing the values
  - `myRow(0).asInstanceOf[String] // String`
  - `myRow.getInt(2)`

DataFrame Transformations

- When working with individual DataFrames:
  - We can add rows or columns
  - We can remove rows or columns
  - We can transform a row into a column
  - We can change the order of rows based on the values of columns

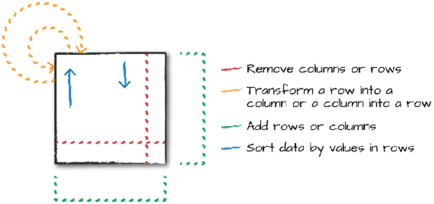


Figure 5-2. Different kinds of transformations

Creating DataFrames

- We can create DataFrames from raw sources
  - Chapter 9 will cover this in more detail
  - We can register raw data as a temporary view
  - Query it with SQL
- We can create a DataFrame on the fly by taking a set of rows and converting them to a DataFrame

Code Example 65

```
df = spark.read.format("json").load("data/flight-data/json/
df.createOrReplaceTempView("dfTable")

import org.apache.spark.sql.Row
import org.apache.spark.sql.types{StructField, StructType,

val myManualSchema = new StructType(Array(new StructField("
val myRows = Seq(Row("Hello", null, 1L))
val myRDD = spark.sparkContext.parallelize(myRows)
val myDf = spark.createDataFrame(myRDD, myManualSchema)
myDf.show()

// use can map Scala Seq directly to DataFrames, but Seq do
val myDf = Seq(("Hello",2,1L)).toDf("col1","col2","col3")

from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, Stri
myManualSchema = StructType([
  StructField("some", StringType(), True),
  StructField("col", StringType(), True),
```

Select and selectExpr

- Use the *select* method when working with columns or expressions
- Use the *selectExpr* method when working with expressions in **strings**
- Both are found in `org.apache.spark.sql.functions`
  - select and selectExpr allow you to execute SQL queries on a DataFrame
  - `df.select("DEST_COUNTRY_NAME").show(2)`
  - `SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2`
- You can select multiple columns by using a comma

```
from pyspark.sql.functions import expr, col, column
df.select(
  expr("DEST_COUNTRY_NAME"),
  col("DEST_COUNTRY_NAME"),
  column("DEST_COUNTRY_NAME")
).show(2)
```

selectExpr

- If you find yourself typing a bunch of *select* then *expr* statements:
  - Then `selectExpr` is the convenient interface you want
  - We can add new columns to a DataFrame
- We can use `selectExpr` to build up complex expressions and create new DataFrames
  - ```
    df.selectExpr("*",("DEST_COUNTRY_NAME = ORIGIN_COUNTRY_N
    ```
  - ```
    SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as w
    ```
- We can specify aggregations over an entire DataFrame
  - ```
    df.selectExpr("avg(count"), "count(distinct(DEST_COUNTRY
    ```
  - ```
    SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FR
    ```

Spark Literals

- Sometimes we need to pass a literal value, such as a constant
  - `from pyspark.sql.functions import lit`
  - `df.selectExpr(expr("*"), lit(1).alias("One")).show(2)`
  - This will come up when you need to check a Row value against a predetermined constant value
- Adding additional columns is possible: `.withColumn()`
  - `df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME`
  - This creates a column with a Boolean if the ORIGIN and DEST Country name match.
    - This can save much time in a lookup later on as you will not have to do String comparison
- Columns can be dropped as well
  - `df.drop("ORIGIN_COUNTRY_NAME").columns`
- You can cast columns as well
  - `df.withColumn("count2", col("count").cast("long"))`
  - Renaming a column is possible using the `.withColumnRenamed("existingColumnName","newColumnName")`

Filter and Where Clauses 72

- In working with Spark DataFrames, you can use both `where` and `filter` on a DataFrame
  - `df.filter(col("count") < 2).show(2)`
  - `df.where("count < 2").show(2)`
  - More details in Chapter 11
- Both statements have the same output, `where` is a familiar term from SQL so the book will use that
- You can chain multiple `where` statements together, Spark will handle the expressions at run time
  -
    ```
    df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAM
    ```
  -
    ```
    SELECT * FROM dfTable WHERE count < 2 AND ORIGIN_COUNTRY
    ```
- You can access distinct results as we saw earlier in the chapter
  -
    ```
    df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").di
    ```

Random Samples and Splits

- Sometimes you want to select a random sample of data for running a test on a small representative set
  - You can use the `sample` method on a DataFrame
  - ```
    seed = 5
    withReplacement = False
    fraction = 0.5
    df.sample(withReplacement, fraction, seed).count()
    ```
- You can split a DataFrame
  - The `seed` definition is how the random selection is begun
  - ```
    dataFrames = df.randomSplit([0.25, 0.75], seed)
    dataFrames[0].count() > dataFrames[1].count()
    ```

Concatenating and Appending Rows (Union)

- Previously we learned that DataFrames are **immutable**
  - How then can we append to a DataFrame?
  - In order to append to a DataFrame, you must **union** the original DataFrame along with the new DataFrame
  - Both DataFrames need to have the same schema and number of columns, otherwise the operation fails
  - “‘python from pyspark.sql import Row schema = df.schema newRows = [Row(“New Country”, “Other Country”, 5L), Row(“New Country 2”, “Other Country 3”, 1L)] paralleizedRows = spark.sparkContext.parallelize(newRows) newDF = spark.createDataFrame(parallelizedRows, schema)

```
- ```python
df.union(newDF).where("count = 1").where(col("ORIGIN_CO
```

Conclusion

- Conclusion here

Questions

- Any questions?
- Read Chapter 06 and do any exercises in the book.