# Spark DSL and SQL

Spark DSL is Domain Specific Language

## Transformation:

### select(columns):

val df = spark.read

.option("inferSchema","true")

.format("csv")

.load("file:D/BigData/spark-essentials-master/src/main/resources/data/movies.csv")

*val df = spark.read.option("inferSchema","true").option("header","true").format("csv").load("file:///D:/BigData/spark-essentials-master/src/main/resources/data/movies.csv")*

df.select('actor, 'year).show(5)

df.select('actor, ('year - ('year %10)).as('decade)).show(5)

### selectExpr(columns):

Select \* ==> df.selectExpr("\*").show(5)

df.selectExpr("title as movie\_name").show(5)

df.selectExpr("title as movie\_name").show(5)

### filler(condition)

### where(condition)

It is used to filter out the rows that don’t meet the given condition

filter ==> returns only the rows that meet the specified condition

df.filter('year < 2000)

df.filter('year === 2000).show()

df.filter('year =!= 2000).show

### distinct, dropDuplicates

dropDuplicates allows you to control which columns should be used in deduplication logic.

If none is specified, the deduplication logic will use all the columns in the DataFrame

df.select("title").distinct.selectExpr("count(title) as movies").show

df.dropDuplicates("title").selectExpr("count(title) as movies").show

### withColumn(colName, column)

add a new column to a DataFrame. It requires two input parameters: a column name and a value in the form of a column expression

df.printSchema()

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

df.withColumn("decade", ('year - 'year % 10)).show(5)

+-----------------+-------------+----+------+

| actor| title|year|decade|

+-----------------+-------------+----+------+

|McClure, Marc (I)|Freaky Friday|2003| 2000|

|McClure, Marc (I)| Coach Carter|2005| 2000|

|McClure, Marc (I)| Superman II|1980| 1980|

|McClure, Marc (I)| Apollo 13|1995| 1990|

|McClure, Marc (I)| Superman|1978| 1970|

+-----------------+-------------+----+------+

scala> val df1 = df.withColumn("decade", ('year - 'year % 10))

df1: org.apache.spark.sql.DataFrame = [actor: string, title: string ... 2 more fields]

scala> df.printSchema

root

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

scala> df1.printSchema

root

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

|-- decade: integer (nullable = true)

------------------------------------------------------------------------------------------------------

### withColumnRenamed(existingColName, newColName)

------------------------------------------------------------------------------------------------------

This transformation is strictly about renaming an existing column name in a DataFrame

To rename a cryptic column name to a more human-friendly nam

Before joining two DataFrames that happen to have one or more same column name

nOte:Notice that if the provided existingColName doesn’t exist in the schema, Spark doesn’t throw an error, and it will silently do nothing

df1.withColumnRenamed("actor","hero")

// renaming a column

val carsWithColumnRenamed = carsDF.withColumnRenamed("Weight\_in\_lbs", "Weight in pounds")

### drop(columnName1, columnName2)

------------------------------------------------------------------------------------------------------

drops the specified columns from the DataFrames

Dropping Two Columns: One Exists and the Other One Doesn’t

scala> df3.printSchema

root

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

scala> val df4 = df3.drop("actor","popular")

df4: org.apache.spark.sql.DataFrame = [title: string, year: int]

scala> df4.printSchema

root

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

## Working with Missing Data or Bad Data

There are three common ways of dealing with missing or bad data

* Drop the rows that have missing values in a one or more columns
* Fill those missing values with user-provided values
* Replace the bad data with something that you know how to deal with

Dropping Rows with Missing Data:

Creating a dataframe with null values

scala> val badMovies = Seq(Row(null, null, null), Row(null, null, 2018L), Row("John Doe", "Awesome Movie", null), Row(null, "Awesome Movie", 2018L), Row("Mary Jane", null, 2018L))

badMovies: Seq[org.apache.spark.sql.Row] = List([null,null,null], [null,null,2018], [John Doe,Awesome Movie,null], [null,Awesome Movie,2018], [Mary Jane,null,2018])

Converting to RDD:

scala> val badMoviesRDD = spark.sparkContext.parallelize(badMovies)

Creating a Schema

scala> val movies = StructType(Array(StructField("actor",StringType,true),StructField("movie",StringType,true),StructField("year",LongType,true)))

Creating a DataFrame:

val badMoviesDF = spark.createDataFrame(badMoviesRDD, movies)

badMoviesDF.show

+---------+-------------+----+

| actor| movie|year|

+---------+-------------+----+

| null| null|null|

| null| null|2018|

| John Doe|Awesome Movie|null|

| null|Awesome Movie|2018|

|Mary Jane| null|2018|

+---------+-------------+----+

// dropping rows that have missing data in any columnRemove:

// both of the lines below will achieve the same purpose

scala> badMoviesDF.na.drop().show

+-----+-----+----+

|actor|movie|year|

+-----+-----+----+

+-----+-----+----+

scala> badMoviesDF.na.drop("any").show

+-----+-----+----+

|actor|movie|year|

+-----+-----+----+

+-----+-----+----+

// drop rows that have missing data in every single column 🡪 Drop row only if all the columns are null

scala> badMoviesDF.na.drop("all").show

+---------+-------------+----+

| actor| movie|year|

+---------+-------------+----+

| null| null|2018|

| John Doe|Awesome Movie|null|

| null|Awesome Movie|2018|

|Mary Jane| null|2018|

+---------+-------------+----+

//Check null values by specifying the column name

scala> badMoviesDF.na.drop(Array("actor")).show

+---------+-------------+----+

| actor| movie|year|

+---------+-------------+----+

| John Doe|Awesome Movie|null|

|Mary Jane| null|2018|

+---------+-------------+----+

## SQL tables

Temp SQL tables can be created on top of DF and DS, for these scope of the table remains as long as the SparkSession, to handle the dataframe as table with SQL queries.

scala> val dfcsv1 = spark.read.format("csv").

| option("mode","dropmalformed").

| option("inferSchema",true).

| load("file:///E:/temp/data/custs").toDF("id","fname","lname","age","prof")

dfcsv1: org.apache.spark.sql.DataFrame = [id: int, fname: string ... 3 more fields]

scala> dfcsv1.printSchema

root

|-- id: integer (nullable = true)

|-- fname: string (nullable = true)

|-- lname: string (nullable = true)

|-- age: integer (nullable = true)

|-- prof: string (nullable = true)

dfcsv1.createOrReplaceTempView("customers")

scala> spark.sql("select \* from customers limit 2").show()

+-------+--------+-----+---+-------+

| id| fname|lname|age| prof|

+-------+--------+-----+---+-------+

|4000001|Kristina|Chung| 55| Pilot|

|4000002| Paige| Chen| 77|Teacher|

+-------+--------+-----+---+-------+

scala> spark.sql("select id, fname from customers where trim(fname) like '%Pet%' limit 2").show

+-------+-----+

| id|fname|

+-------+-----+

|4000387|Peter|

|4000456|Peter|

+-------+-----+

// mixing SQL statement and DataFrame transformation

scala> spark.sql("select distinct(fname) as customers\_names, age from customers ").where('age > 30).orderBy('age.desc).show(3)

+---------------+---+

|customers\_names|age|

+---------------+---+

| Paige| 77|

| Floyd| 75|

| Nancy| 75|

+---------------+---+

// using a subquery

scala> spark.sql("""select distinct(fname), age from

| (select id,fname,lname,age from customers where age = 35)""").show(3)

+------+---+

| fname|age|

+------+---+

| Ian| 35|

|Marion| 35|

| Kevin| 35|

+------+---+

## Writing Data Out

Basic Syntax:

movies.write.format(...).mode(...).option(...).partitionBy(...).bucketBy(...) .sortBy(...).save(path)

Default format is Parquet

The partitionBy, bucketBy, and sortBy functions are used to control the directory structure of the output files in the file-based data sources

save mode, which controls how Spark will handle the situation when the specified output folder already exists

|  |  |
| --- | --- |
| **Mode** | **Description** |
| append | This appends the DataFrame data to the list of files that already exist at the specified destination location |
| overwrite | This completely overwrites any data files that already exist at the specified destination location with the data in the DataFrame |
| **error errorIfExists (default)** | This is the default mode. If the specified destination location exists, then DataFrameWriter will throw an error. |
| ignore | If the specified destination location exists, then simply do nothing. In other words, silently don’t write out the data in the DataFrame |

val dfcsv1 = spark.read.format("csv").option("mode","dropmalformed").option("inferSchema",true).load("file:///E:/temp/data/custs").toDF("id","fname","lname","age","prof")

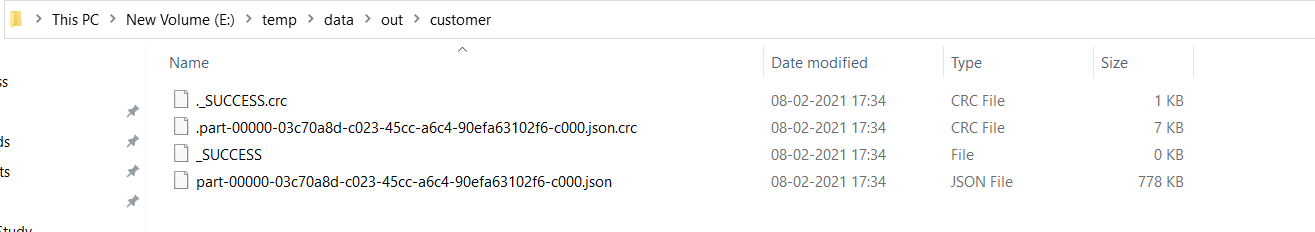
val cleandata = dfcsv1.na.drop()

cleandata.write.format("csv").mode("overwrite").save("file:///E:/temp/data/out/customer")

cleandata.write.format("json").mode("overwrite").save("file:///E:/temp/data/out/customer")

cleandata.write.mode("overwrite").save("file:///E:/temp/data/out/customer") //write in parquet file

output folder:

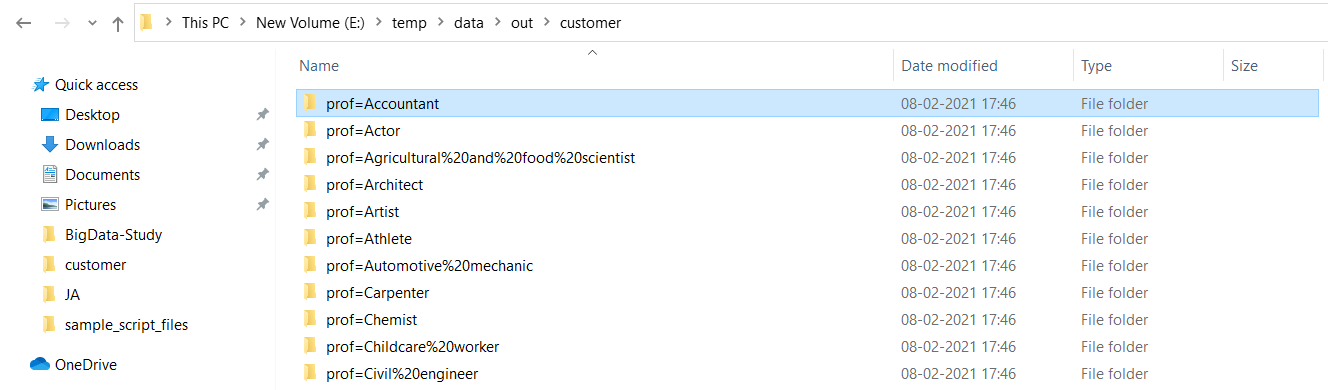


#### Write using partitionby

cleandata.write.format("csv").option("header",true).mode("overwrite").partitionBy("prof").save("file:///E:/temp/data/out/custom

er")

output folder now looks like



## DataFrame Persistence

Spark SQL knows the schema of the data inside a DataFrame, so it organizes the data in a columnar format as well as applies any applicable compressions to minimize space usage. The net result is it will require much less space to store a DataFrame in memory than storing an RDD when both are backed by the same data file

After cache Dataframe occupies less memory than RDD having same data

spark.catalog.cacheTable("num\_df")

## DataFrame Joins

Performing a join will combine the columns of two datasets (could be different or same), and the combined DataFrame will contain columns from both sides

Creating Two Small DataFrames:

case class Employee(first\_name:String, dept\_no:Long)

val employeeDF = Seq( Employee("John", 31),

Employee("Jeff", 33),

Employee("Mary", 33),

Employee("Mandy", 34),

Employee("Julie", 34),

Employee("Kurt", null.

asInstanceOf[Int])

).toDF

case class Dept(id: Long, name: String)

val deptDF = Seq(

Dept(31, "Sales"),

Dept(33, "Engineering"),

Dept(34, "Finance"),

Dept(35, "Marketing")

).toDF

employeeDF.createOrReplaceTempView("employees")

deptDF.createOrReplaceTempView("departments")

### Inner Join

spark.sql("select \* from employees JOIN departments on dept\_no == id").show

or

// define the join expression of equality comparison

val deptJoinExpression = employeeDF.col("dept\_no") === deptDF.col("id")

// perform the join

employeeDF.join(deptDF, joinExpression, "inner").show

or

employeeDF.join(deptDF).where('dept\_no === 'id).show

### Left Outer Joins

spark.sql("select \* from employees left outer join departments ON employees.dept\_no == departments.id").show

### Right Outer Joins

spark.sql("select \* from employees right outer join departments ON employees.dept\_no == departments.id").show

### Outer Joins (aka Full Outer Joins)

spark.sql("select \* from employees full outer join departments on employees.dept\_no == departments.id").show

### Cross (aka Cartesian)

spark.sql("select \* from employees CROSS JOIN departments").show(50)

## User Defined Functions (UDFs)

UDFs are just regular functions that take some inputs and provide an output

UDFs must be registered with Spark before they are used so Spark knows to ship them to executors to be used and executed

**STEPs**:

1. Write a function and test it
2. Register that function with Spark by passing in the function name and its signature to Spark’s udf function
3. Use UDF in either the DataFrame code or when issuing SQL queries

// create student grades DataFrame

case class Student(name: String, score: Int)

val studentDF =

Seq(Student("Joe", 85), Student("Jane", 90), Student("Mary", 55)).toDF()

// register as a view

studentDF.createOrReplaceTempView("students")

// create a function to convert grade to letter grade

def letterGrade(score: Int): String = {

score match {

case score if score > 100 => "Cheating"

case score if score >= 90 => "A"

case score if score >= 80 => "B"

case score if score >= 70 => "C"

case \_ => "F"

}

}

// register as a UDF

val letterGradeUDF = udf(letterGrade(\_: Int): String)

// use the UDF to convert scores to letter grades

studentDF.select($"name", $"score", letterGradeUDF($"score").as("grade")).show

scala> spark.sql("select name, score, letterGrade(score) as grade from students").show

+----+-----+-----+

|name|score|grade|

+----+-----+-----+

| Joe| 85| B|

|Jane| 90| A|

|Mary| 55| F|

+----+-----+-----+