Spark Program

CHAPTER 2: DATAFRAMES

Chapter Objectives

In this chapter, we will:

- Introduce DataFrames
- Show how to create a structured object using DataFrames
- Apply transformations and actions on DataFrames

DataFrames

- Spark 2.0 introduced a more feature rich and easier to use version of RDD’s known as a DataFrame
  - Modeled to be similar to Pandas DataFrame so it is easily familiar
  - Is an RDD but has column names and data types
  - Has transformations and actions that are easier to use than RDD versions
  - Attempts to be more SQL-like for even more familiarity
  - Can read and write many more file formats than basic RDD’s could

Make a DataFrame

- Spark 2.0 introduced the `SparkSession`, simply called spark in PySpark
  - Provides easier access to the different spark contexts
  - `spark.sparkContext` is the same as the old `sc`
- Once we have the Spark context, we can start using DataFrames

```
x = sc.parallelize([(1,'alpha'),(2,'beta')])
x0 = spark.createDataFrame(x)
x0.show()

+---+-----+
| _1|   _2|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+
```

Column Names

- To make the DataFrame more useful, column names can be applied

```
x1 = spark.createDataFrame(x, schema = ['ID', 'Name'])
x1.show()
x1.describe()

+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[summary: string, ID: string, Name: string]
```

Schemas

- To make the DataFrame even more useful, a schema with data types can be applied

```
x2 = spark.createDataFrame(x, 'ID:int, Name:string')
x2.show()
print(x2)
+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[ID: int, Name: string]
```

Schema Objects

- Sometimes a schema object is required or just preferred

```
schema = StructType([
    StructField('ID', IntegerType()),
    StructField('Name', StringType())
])
x3 = spark.createDataFrame(x, schema = schema)
x3.show()
print(x3)
```

```
+---+-----+
| ID| Name|
+---+-----+
|  1|alpha|
|  2| beta|
+---+-----+

DataFrame[ID: int, Name: string]
```

`toDF()` Method

- Alternatively, RDD's have a `toDF()` method which works the same as `spark.createDataFrame()`

```
x.toDF()

x.toDF(['ID', 'Name'])

x.toDF('ID:int, Name:string')

x.toDF(schema = schema1)
```

Reading Files

- There are many file formats directly supported for reading and writing
  - `csv`
  - `json`
  - `orc`
  - `parquet`
  - `jdbc`
- Other formats can be loaded using custom Java classes
  - Cassandra
  - Mongo
  - HBase
  - AVRO

Reading CSV Files

- There are many different syntaxes that you will see but they all do the same thing
- The `sep` parameter can also be used to indicate different separators like `\t` for tab

```
filename = '/class/datasets/finance/CreditCard.csv'

df4 = spark.read.load(filename, format = 'csv',
  sep = ',', inferSchema = True, header = True)

df4 = spark.read.format('csv').option('header','true').
  option('inferSchema','true').load(filename)

df4 = spark.read.csv(filename, header = True,
  inferSchema = True)
```

Writing Files

- The write method on a DataFrame can be used just like the `read` function using many different options
- Some options are built-in such as:
  - `jdbc`
  - `json    spark.read.json(filename) df.write.json(file)`
  - `orc     spark.read.orc(filename)  df.write.orc(file)`
  - `parquet spark.read.parquet(file)  df.write.parquet(file)`
  - `text    spark.read.text(file)     df.write.text(file)`
- Other formats can use the option to supply a custom Java class that can be downloaded and installed on the computer
  - AVRO:
    `spark.read.format("com.databricks.spark.avro").load("kv.avro")`
  - Cassandra:
    `sqlContext.read.format("org.apache.spark.sql.cassandra").options(table = table_name, keyspace = keys_space_name).load()`

Selecting Columns

- DataFrames have methods with names similar to SQL commands

```
prod.select('productid', 'productname', 'unitprice')

p.select(p.productid, p.productname).distinct()

p.select(p.categoryid).distinct()

prod.sort(prod.unitprice)

prod.orderBy('unitprice', ascending = False)

p.select('productid', 'productname', 'unitprice') \
    .orderBy('unitprice')
```

Sorting

- The `sort` and `orderBy` methods are different aliases for the same function

```
p.sort(p.categoryid)
```

- They sort a DataFrame in ascending order or descending order if you pass the `ascending = False` parameter

```
p.sort(p.unitprice, ascending = False)
```

- You can sort on multiple columns

```
p.orderBy('categoryid', 'productid')
```

- Custom sort functions can use the `withColumn` method

Calculated Columns

- New columns can be added to a DataFrame

```
prod2 = prod.withColumn('value', prod.unitprice *
prod.unitsinstock)
```

- Columns can be removed when not needed

```
prod2 = prod2.drop('quantityperunit')
```

Filtering Data

- DataFrames can be filtered like a SQL table using either the `filter` or `where` method
  - They are the exact same method with different aliases

```
p.filter(p.unitprice > 100)

p.filter('unitprice > 100')

p.where(p.categoryid == 2)

p.where('categoryid = 2')

p.where('unitprice >=50 and unitprice <= 100'))

p.where('unitprice between 50 and 100')

p.where((p.unitprice >=50) & (p.unitprice <= 100))
```

JOIN

- DataFrames can be joined to other DataFrames just as you would in SQL and all the expected types are supported
  - INNER
  - LEFT
  - RIGHT
  - FULL

```
tab1 = sc.parallelize([(1, 'Alpha'), (2, 'Beta'), (3,
'Delta')]).toDF('ID:int, code:string')

tab2 = sc.parallelize([(100, 'One', 1), (101, 'Two', 2),
(102, 'Three', 1), (103, 'Four', 4)]) \
.toDF('ID:int, name:string, parentID:int')

tab1.join(tab2, tab1.ID == tab2.parentID).show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'left').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'right').show()
tab1.join(tab2, tab1.ID == tab2.parentID, 'full').show()
```

Grouping and Aggregating

- Grouping in Spark works a little differently—the `groupBy` method creates a grouped DataFrame which can then have aggregate methods called on it

```
tab3 = sc.parallelize([(1, 10), (1, 20), (1, 30), (2,
40), (2,50)]).toDF('groupID:int, amount:int')
x = tab3.groupby('groupID')
```

- There are various different syntaxes to accomplish the same results
  – Call the method after grouping
    ```
    x.max().show()
    ```
  – Use the `agg` method with a dictionary
    ```
    x.agg({'amount':'sum', 'amount':'max'}).show()
    ```
  – Use the `agg` method with the function names
    ```
    from pyspark.sql import functions as F
    x.agg(F.sum('amount'), F.max('amount')).show()
    ```

Miscellaneous Useful Methods

- A lot of standard SQL is supported by Spark
- Using the `expr` function in combination with `withColumn` you can add calculated columns to a DataFrame if you can code the calculation as standard SQL

```
from pyspark.sql.functions import expr
x2.withColumn('uppername', expr('upper(name)')).show()
```

- Sometimes you just want to easily rename a column
  - `withColumnRenamed(oldname, newname)`
- Like `collect()` the `toLocalIterator()` method will return all the results to the driver node, but it does it as a generator instead of a `list`
- Just like SQL there are methods `union, unionAll, subtract`, and `intersect`

Chapter Summary

In this chapter, we have:

- Introduced DataFrames
- Shown how to create a structured object using DataFrames
- Applied transformations and actions on DataFrames