COMP9313: Big Data Management

Lecturer: Xubo Wang

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 5.1: Spark III

Part 1: Spark Structured APIs

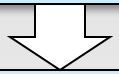
A Brief Review of RDD

- The RDD is the most basic abstraction in Spark. There are three vital characteristics associated with an RDD:
  - Dependencies (lineage)
    - When necessary to reproduce results, Spark can recreate an RDD from the dependencies and replicate operations on it. This characteristic gives RDDs resiliency.
  - Partitions (with some locality information)
    - Partitions provide Spark the ability to split the work to parallelize computation on partitions across executors
    - Reading from HDFS—Spark will use locality information to send work to executors close to the data
  - Compute function: Partition => Iterator[T]
    - An RDD has a compute function that produces an Iterator[T] for the data that will be stored in the RDD.

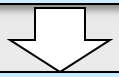
Compute Average Values for Each Key

- Assume that we want to aggregate all the ages for each name
  - group by name, and then compute the average age for each name

```
pairs = sc.parallelize([("Brooke", 20), ("Denny", 31),
("Jules", 30),("TD", 35), ("Brooke", 25)])
```

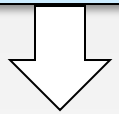


```
pairs1 = pairs.mapValues(lambda x: (x, 1))
```



```
pairs2 = pairs1.reduceByKey(lambda x, y: (x[0] + y[0], x[1]+y[1]))
```

```
avg = pairs2.mapValues(lambda x: x[0]/x[1]).collect()
```



```
[('Brooke', 22.5), ('Denny', 31.0), ('TD', 35.0), ('Jules', 30.0)]
```

Problems of RDD Computation Model

- The compute function (or computation) is opaque to Spark
  - Whether you are performing a join, filter, select, or aggregation, Spark only sees it as a lambda expression

```
pairs1 = pairs.mapValues(lambda x: (x, 1))
```

- Spark has no way to optimize the expression, because it's unable to inspect the computation or expression in the function.
- Spark has no knowledge of the specific data type in RDD
  - To Spark it's an opaque object; it has no idea if you are accessing a column of a certain type within an object

Spark's Structured APIs

- Spark 2.x introduced a few key schemes for structuring Spark,
- This specificity is further narrowed through the use of a set of common operators in a DSL (domain specific language), including the Dataset APIs and DataFrame APIs
  - These operators let you tell Spark what you wish to compute with your data
  - It can construct an efficient query plan for execution.
- Structure yields a number of benefits, including better performance and space efficiency across Spark components

Spark's Structured APIs

- E.g, for the average age problem, using the DataFrame APIs:

```
from pyspark.sql.functions import avg

dataDF = spark.createDataFrame([("Brooke", 20),
("Denny", 31), ("Jules", 30),("TD", 35), ("Brooke",
25)], schema = 'name string, age int')

dataDF.groupBy("name").agg(avg("age")).show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

Spark's Structured APIs

- Spark now knows exactly what we wish to do: group people by their names, aggregate their ages, and then compute the average age of all people with the same name.

- Spark can inspect or parse this query and understand our intention, and thus it can optimize or arrange the operations for efficient execution.

Datasets and DataFrames

- A *Dataset* is a distributed collection of data
  - It provides the benefits of RDDs (e.g., strong typing) with the benefits of Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects (only available in Scala and Java) and then manipulated using functional transformations (map, flatMap, etc.)

- A *DataFrame* is a *Dataset* organized into named columns
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
  - An abstraction for selecting, filtering, aggregating and plotting structured data
  - A DataFrame can be represented by a Dataset of Rows
    - Scala: DataFrame is simply a type alias of Dataset[Row]
    - Java: use Dataset<Row> to represent a DataFrame

DataFrame API

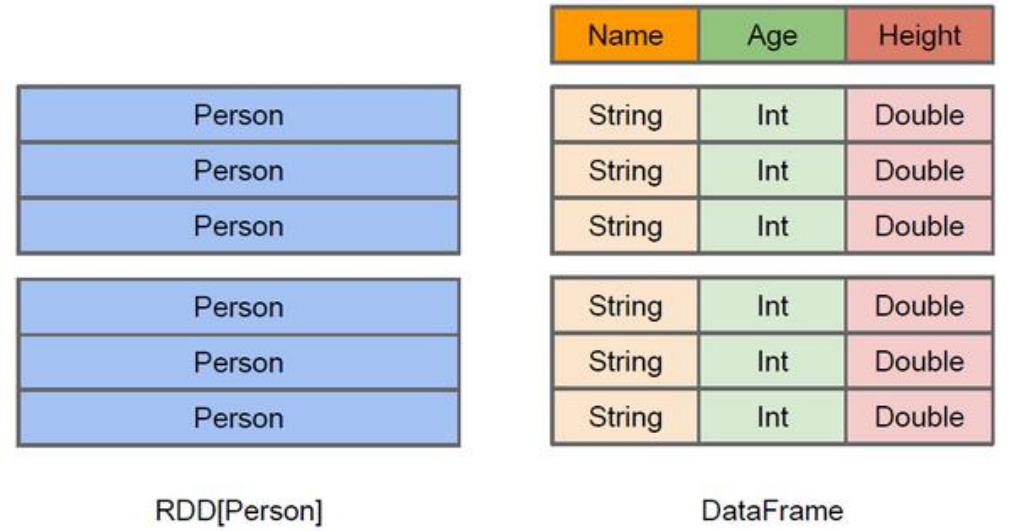
- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type.
- When data is visualized as a structured table, it's not only easy to digest but also easy to work with

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]

The table-like format of a DataFrame

Difference between DataFrame and RDD

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus, Spark SQL query optimizer can target optimization

DataFrame Data Sources

- Spark SQL's Data Source API can read and write DataFrames using a variety of formats.
  - E.g., structured data files, tables in Hive, external databases, or existing RDDs

Create DataFrames with Schema

- A PySpark DataFrame can be created via pyspark.sql.SparkSession.createDataFrame typically by passing a list of lists, tuples, dictionaries and pyspark.sql.Rows, a pandas DataFrame and an RDD consisting of such a list.
- You can Create a PySpark DataFrame with an explicit schema.

```
// Given a list of pairs including names and ages
data = [("Brooke", 20), ("Denny", 31), ("Jules", 30),("TD", 35),
("Brooke", 25)]

// Create DataFrame' from a list
dataDF = spark.createDataFrame(data, schema = "name string, age
int")
```

- You can create a PySpark DataFrame from an RDD consisting of a list of tuples.

```
// Given a pair RDD including name and age
data = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25)])

// Create DataFrame' from an RDD
dataDF = spark.createDataFrame(data, schema=["name","age"])
```

Create DataFrames with Schema

- You can also create a PySpark DataFrame from a list of rows

```
from pyspark.sql import Row

// Given a list of rows containing names and ages
data = [Row(name = "Brooke", age = 20), Row(name = "Denny", age =
31), Row(name = "Jules", age = 30), Row(name = "TD", age = 35),
Row(name = "Brooke", age = 25)]

// Create DataFrame' from a list of Rows
dataDF = spark.createDataFrame(data)
```

- You can print out the schema of a DataFrame

```
>>> dataDF.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
```

- All the DataFrame APIs are listed here: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/dataframe.html

show()

- The top rows of a DataFrame can be displayed using DataFrame.show().

- PySpark DataFrame show() is used to display the contents of the DataFrame. By default, it shows only 20 Rows, and the column values are truncated at 20 characters.

```
>>> dataDF.show()
+------+---+
|  name|age|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

Schemas in Spark

- A schema in Spark defines the column names and associated data types for a DataFrame
- Defining a schema up front offers three benefits
  - You relieve Spark from the onus of inferring data types.
  - You prevent Spark from creating a separate job just to read a large portion of your file to ascertain the schema, which for a large data file can be expensive and time-consuming.
  - You can detect errors early if data doesn't match the schema.
- Define a DataFrame programmatically with three named columns, author, title, and pages

```
from pyspark.sql.types import *
schema = StructType([StructField(”title", StringType(), False),
StructField(”author", StringType(), False),
StructField("pages", IntegerType(), False)])
```

StructType & StructField

- PySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and create complex columns like nested struct, array, and map columns.
- StructType is a collection of StructField's that defines column name, column data type, boolean to specify if the field can be nullable or not and metadata.

```
from pyspark.sql.types import *
schema = StructType([StructField("title", StringType(), False),
StructField("author", StringType(), False),
StructField("pages", IntegerType(), False)])
#schema = "title string, author string, pages int"

data = [("book1", "author1",300), ("book2", "author2",600),
("book3", "author3",900)]

dataDF = spark.createDataFrame(data, schema)
```

```
>>> dataDF.printSchema()
root
 |-- title: string (nullable = true)
 |-- author: string (nullable = true)
 |-- pages: integer (nullable = true)
```

```
>>> dataDF.show()
+-----+-------+-----+
|title| author|pages|
+-----+-------+-----+
|book1|author1|  300|
|book2|author2|  600|
|book3|author3|  900|
+-----+-------+-----+
```

Spark's Basic Data Types (Python)

- Spark supports basic internal data types, which can be declared in your Spark application or defined in your schema

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Structured and Complex Data Types (Python)

- For complex data analytics, you'll need Spark to handle complex data types, such as maps, arrays, structs, dates, timestamps, fields, etc.

Data type	Value assigned in Python	API to instantiate
BinaryType	bytearray	BinaryType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType([fields])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

Complex Data Type Example

- PySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and create complex columns like nested struct, array, and map columns.
- StructType is a collection of StructField's that defines column name, column data type, boolean to specify if the field can be nullable or not and metadata.

```
from pyspark.sql.types import *
data = [("Alice", ["Java","Scala"]), ("Bob", ["Python", "Scala"])]
schema = StructType([
   StructField("name", StringType()),
   StructField("languagesSkills", ArrayType(StringType())),
])
df = spark.createDataFrame(data, schema)
df.printSchema()
```

```
>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- languagesSkills: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

```
>>> df.show()
+-----+---------------+
| name|languagesSkills|
+-----+---------------+
|Alice|  [Java, Scala]|
|  Bob|[Python, Scala]|
+-----+---------------+
```

Reuse Schema

- Simply execute dataDF.schema

```
from pyspark.sql.types import *
schema = StructType([StructField("title", StringType(), False),
StructField("author", StringType(), False),
StructField("pages", IntegerType(), False)])

data = [("book1", "author1",300), ("book2", "author2",600),
("book3", "author3",900)]

dataDF = spark.createDataFrame(data, schema)
```

```
>>> dataDF.schema
StructType([StructField('title', StringType(), False), StructField('author', StringType(), False
), StructField('pages', IntegerType(), False)])
```

Columns

- Each column describes a type of field
- We can list all the columns by their names, and we can perform operations on their values using relational or computational expressions
  - List all the columns

```
>>> dataDF.columns
['name', 'age']
```

  - Access a particular column with its name and it returns a Column type

```
>>> dataDF.name
Column<'name'>
```

  - We can also use logical or mathematical expressions on columns

```
>>> dataDF.select(dataDF.age * 2).show()
+---------+
|(age * 2)|
+---------+
|       40|
|       62|
|       60|
|       70|
|       50|
+---------+
```

Columns

- withColumn() returns a new DataFrame by adding a column or replacing the existing column that has the same name
  - from pyspark.sql.functions import upper
  - dataDF.withColumn("name_upper", upper(dataDF.name)).show()
  - dataDF.withColumn("name", upper(dataDF.name)).show()

```
>>> from pyspark.sql.functions import upper
>>> dataDF.withColumn("name_upper", upper(dataDF.name)).show()
+------+---+----------+
|  name|age|name_upper|
+------+---+----------+
|Brooke| 20|    BROOKE|
| Denny| 31|     DENNY|
| Jules| 30|     JULES|
|    TD| 35|        TD|
|Brooke| 25|    BROOKE|
+------+---+----------+
```

```
>>> dataDF.withColumn("name", upper(dataDF.name)).show()
+------+---+
|  name|age|
+------+---+
|BROOKE| 20|
| DENNY| 31|
| JULES| 30|
|    TD| 35|
|BROOKE| 25|
+------+---+
```

Rows

- A row in Spark is a generic Row object, containing one or more columns
- Row is an object in Spark and an ordered collection of fields, We can access its fields by an index starting at 0
- Row objects can be used to create DataFrames. A DataFrame can be collected as a list of Rows

```
>>> from pyspark.sql import Row
>>> data = [Row(name = "Brooke", age = 20), Row(name = "Denny", age = 31), Row(n
ame = "Jules", age = 30), Row(name = "TD", age = 35), Row(name = "Brooke", age =
 25)]
>>> dataDF = spark.createDataFrame(data)
>>> dataDF.show()
+------+---+
|  name|age|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+

>>> dataDF.collect()
[Row(name='Brooke', age=20), Row(name='Denny', age=31), Row(name='Jules', age=30
), Row(name='TD', age=35), Row(name='Brooke', age=25)]
```

Projection and Filter

- select(): projects a set of expressions and returns a new DataFrame.

```
>>> data_df.select(data_df.name,(data_df.age + 10).alias('age')).show()
+------+---+
|  name|age|
+------+---+
|Brooke| 30|
| Denny| 41|
| Jules| 40|
|    TD| 45|
|Brooke| 35|
+------+---+
```

- filter(): filters rows using the given condition.
  - where() is an alias for filter().

```
>>> data_df.where(data_df.age > 30).show()
+-----+---+
| name|age|
+-----+---+
|Denny| 31|
|   TD| 35|
+-----+---+
```

```
>>> data_df.filter(data_df.age > 30).show()
+-----+---+
| name|age|
+-----+---+
|Denny| 31|
|   TD| 35|
+-----+---+
```

Grouping Data

- DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition, applies a function to each group and then combines them back to the DataFrame.

```
>>> df.show()
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

  - Grouping and then applying the avg() function to the resulting groups

```
>>> df.groupby('color').avg().show()
+-----+-------+-------+
|color|avg(v1)|avg(v2)|
+-----+-------+-------+
|  red|    4.8|   48.0|
|black|    6.0|   60.0|
| blue|    3.0|   30.0|
+-----+-------+-------+
```

Spark's Structured APIs

- E.g, for the average age problem, using the DataFrame APIs:

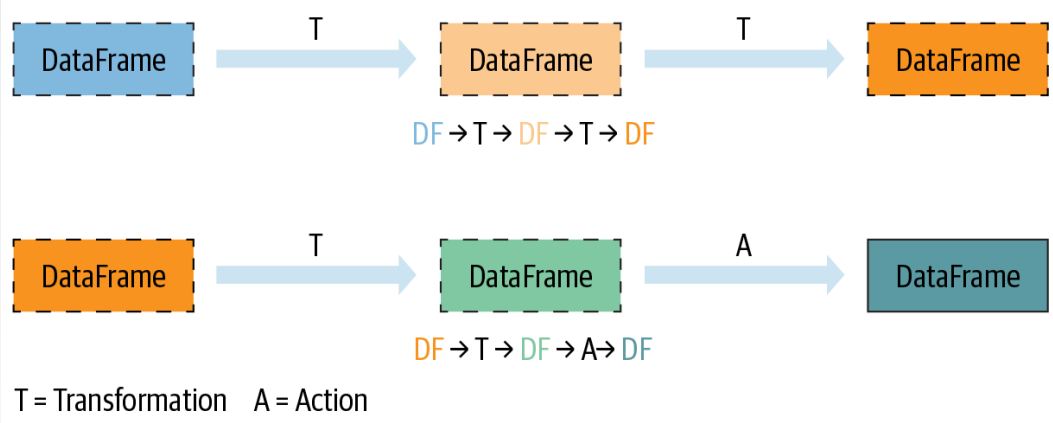
```
from pyspark.sql.functions import avg

data_df = spark.createDataFrame([("Brooke", 20),
("Denny", 31), ("Jules", 30),("TD", 35), ("Brooke",
25)], schema = 'name string, age int')

data_df.groupBy("name").avg().show()
#data_df.groupBy("name").agg(avg("age")).show()
#data_df.groupBy("name").agg({"age":"avg"}).show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

- All the Grouping APIs are listed here: https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/grouping.html

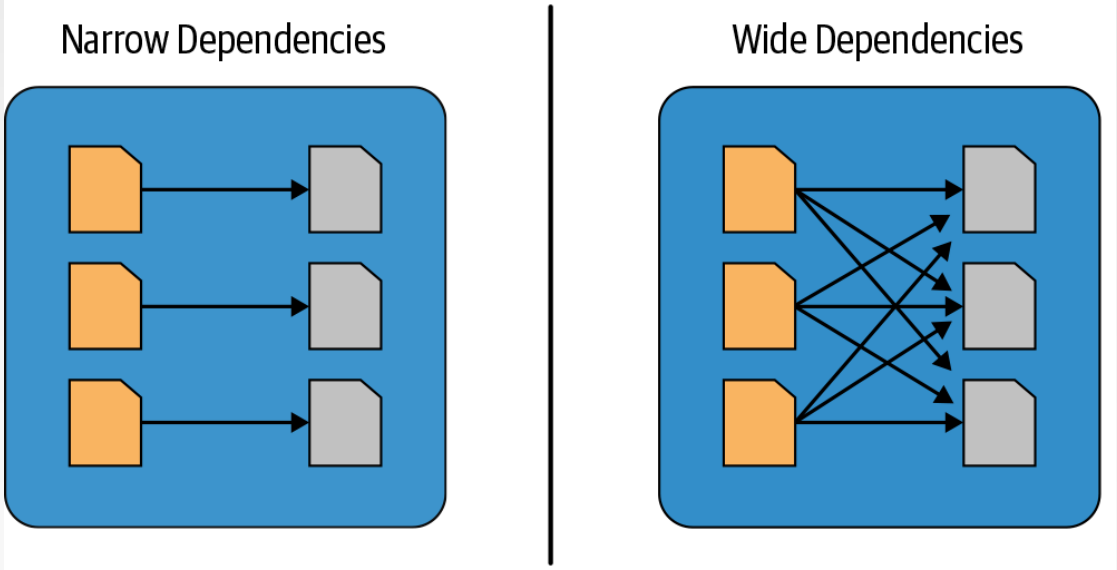
Transformations, Actions, and Lazy Evaluation

- Spark DataFrame operations can also be classified into two types: transformations and actions.
  - All transformations are evaluated lazily - their results are not computed immediately, but they are recorded or remembered as a lineage
  - An action triggers the lazy evaluation of all the recorded transformations



Narrow and Wide Transformations

- Transformations can be classified as having either narrow dependencies or wide dependencies
  - Any transformation where a single output partition can be computed from a single input partition is a narrow transformation, like filter()
  - Any transformation where data from other partitions is read in, combined, and written to disk is a wide transformation, like groupBy()



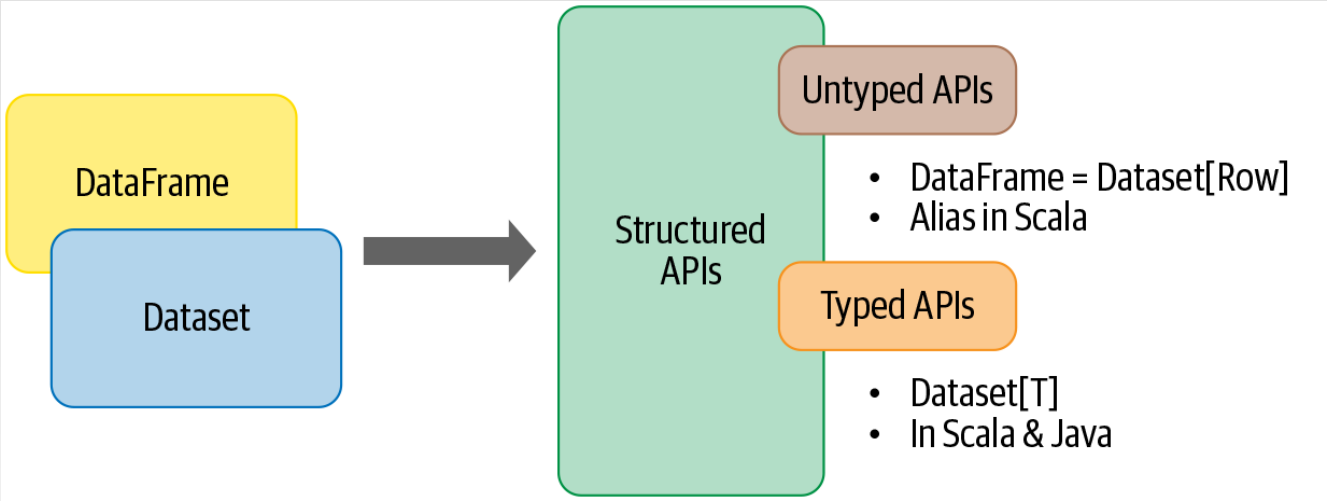
WordCount using DataFrame

```
fileRDD =
spark.sparkContext.textFile("file:///home/comp9313/inputText")

wordsDF = fileRDD.flatMap(lambda x: x.split(" ")).map(lambda x:
(x, )).toDF("word string")


#or
# from pyspark.sql.types import StringType
# wordsDF = spark.createDataFrame(fileRDD.flatMap(lambda x:
x.split(" ")), StringType()).withColumnRenamed("value", "word")

countDF = wordsDF.groupBy("word").count()

countDF.show()

countDF.sort(desc("count")).show()
```

```
>>> countDF = wordsDF.groupBy(wordsDF.word).count()
>>> countDF.show()
+------+-----+
|  word|count|
+------+-----+
| World|    1|
| Hello|    2|
|   Bye|    1|
|Hadoop|    2|
+------+-----+
```

DataSet

- Spark 2.0 unified the DataFrame and Dataset APIs as Structured APIs with similar interfaces



- The Datasets are *strong typed*, and so the typed errors can be detected during compile-time, providing better type safety and reliability.
- In most cases, DataFrames are sufficient unless you have special requirements or a need for stronger type safety.

DataFrames/Datasets vs. RDD

- If you want to tell Spark what to do, not how to do it, use DataFrames or Datasets.
- If you want rich semantics, high-level abstractions, and DSL operators, use DataFrames or Datasets.
- If your processing demands *high-level* expressions, filters, maps, aggregations, computing averages or sums, SQL queries, columnar access, or use of relational operators on semi-structured data, use DataFrames or Datasets.
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames.
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames.
- If you want space and speed efficiency, use DataFrames.
- If you are a Python user, use DataFrames and drop down to RDDs if you need more control.
  - RDD is not deprecated!

Part 2: Running on a Cluster

WordCount (RDD)

- Standalone code

```
from pyspark import SparkContext, SparkConf

conf = SparkConf()
conf.setMaster("local").setAppName("wordcount")
sc = SparkContext(conf=conf)

#Or you can do the below directly
#sc = SparkContext('local', wordcount')

text = sc.textFile("text.txt")
count = text.flatMap(lambda line: line.split()).map(lambda word: (word,
1)).reduceByKey(lambda a, b : a + b)
count.saveAsTextFile("results")
sc.stop()
```

  - Refer to the document of SparkConf and SparkContext
- Use spark-submit to run the code in a cluster:
  - spark-submit wordcount.py

WordCount (DataFrame)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark =
SparkSession.builder.appName("WordCount").master("local").getOrCreate()

fileDF = spark.read.text("text.txt")

wordsDF = fileDF.select(explode(split(fileDF.value, " ")).alias("word"))

countsDF = wordsDF.groupBy("word").agg(count("*").alias("count"))

countsDF.write.format("csv").save("results")

spark.stop()
```

Deploying Applications in Spark

- spark-submit

Common flags	Explanation
--master	Indicates the cluster manager to connect to
--name	A human-readable name for your application. This will be displayed in Spark's web UI.
--executor-memory	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes)
--driver-memory	The amount of memory to use for the driver process, in bytes.

- spark-submit wordcount.py --master spark://hostname:7077 \
  --name wordcount \
  --executor-memory 2g
  YOUR_Python "options" "go here"

References

- http://spark.apache.org/docs/latest/index.html
- Spark SQL guide: http://spark.apache.org/docs/latest/sql-programming-guide.html
- https://spark.apache.org/docs/3.3.0/api/scala/org/apache/spark/index.html
- https://spark.apache.org/docs/3.3.0/api/python/getting_started/index.html
- Learning Spark. 2nd edition
- Spark SQL Functions: https://sparkbyexamples.com/spark/spark-sql-functions/

End of Chapter 5.1