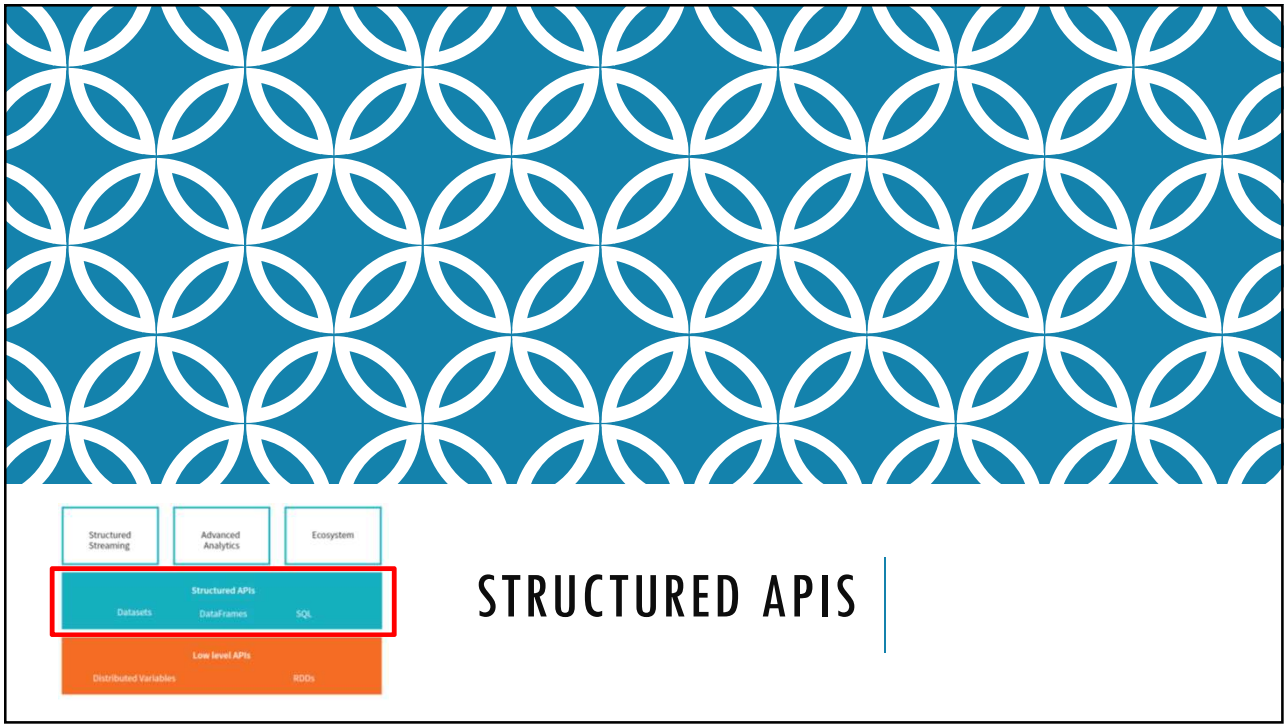


STRUCTURED APIS

1

MAIN REFERENCES

1. Spark: The Definitive Guide
2. Pyspark documentation

2

STRUCTURED APIS

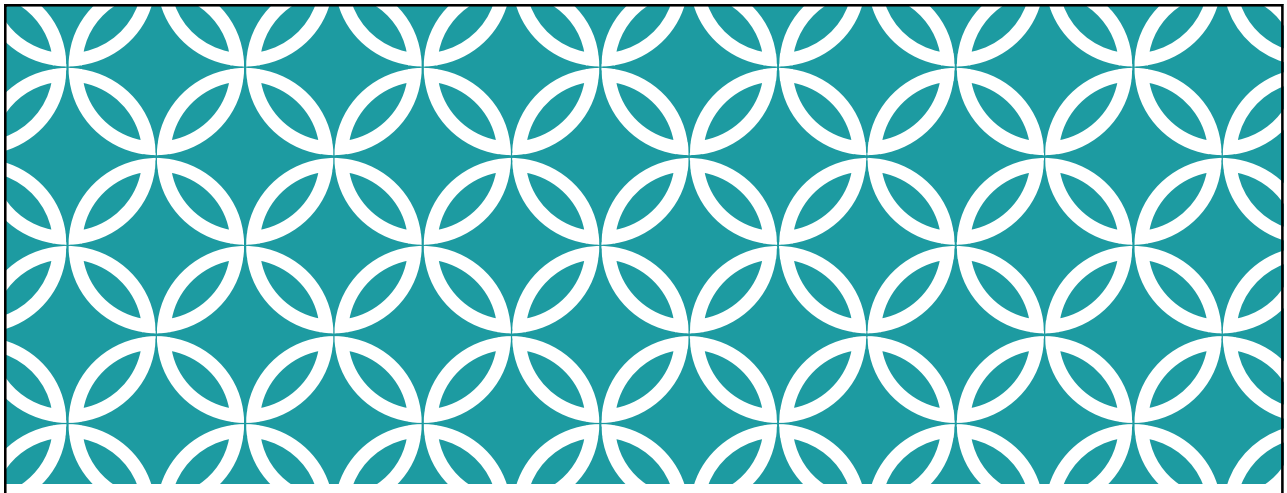
Collection of high-level APIs to manipulate data in Spark

Using these, Spark is able to perform several optimizations

Composed by three elements

- Dataset API
- DataFrame API
- SQL

3



DATAFRAME API

4

DATAFRAME

Abstraction built on top of RDDs to represent immutable distributed data sets that have a ***table-like structure***

Data are organized as a collection of *records* (each one is an instance of *Row*) and *columns,* which define the *fields* that structure the *record's* data

Columns are defined by its *schema,* each one, having a name and a type

5

DATAFRAME

Created from

- a list of its elements (records),
- by reading data from a datasource, or
- by applying transformations on another DataFrame

DataFrame's schema can be inferred from the data, or be specified by the developer

6

CREATING DATAFRAMES WITH SCHEMA INFERENCE

1. Create the instances of objects (registries) that will compose the DataFrame

```
users1 = [(1, 'Fabio', 47), (2, 'Andrea', 47), (3, 'Thiago', 21)]
users2 = [[1, 'Fabio', 47], [2, 'Andrea', 47], [3, 'Thiago', 21]]
users3 = [Row(1, 'Fabio', 47), Row(2, 'Andrea', 47)]

users4 = [{'userId': 1, 'name': 'Fabio', 'age': 47}, {'userId': 2, 'name': 'Andrea', 'age': 47}]
users5 = [Row(userId=1, name='Fabio', age=47), Row(userId=2, name= 'Andrea', age= 47)]

User = Row('userId', 'name', 'age')
user1 = User(1, 'Fabio', 47)
user2 = User(2, 'Andrea', 47)
users6 = [user1, user2]
```

CREATING DATAFRAMES WITH SCHEMA INFERENCE

2. Create a SparkSession

```
spark = SparkSession \
.builder \
.master('local[*]') \
.appName("DataFrames") \
.getOrCreate()
```

3. Use it to create DataFrames

```
df3 = spark.createDataFrame(users3)
df3.printSchema()
print(df3.take(2))

df4 = spark.createDataFrame(users4)
df4.printSchema()
print(df4.take(2))

df5 = spark.createDataFrame(users5)
df5.printSchema()
print(df5.take(2))
```

```
[Row(age=47, name='Fabio', userId=1), Row(age=47, name='Andrea', userId=2)]
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
 |-- userId: long (nullable = true)
```

8

CREATING DATAFRAMES USING USER DEFINED *SCHEMAS*

1. Define a schema
   - (A) programmatically or
   - (B) using a *Data Definition Language (DDL)*
2. Create a DataFrame from a SparkSession

9

1A. DEFINING A SCHEMA PROGRAMMATICALLY

Create an instance of the StructType class from a list of StructFields, each one defining a fied of a row (i.e. a co*lumn*)

```
# Programmatically:
schema1 = StructType([ \
    StructField("userId", IntegerType(), False),
    StructField("name", StringType(), False, {'name':'The user\'s fullname', \
                                              'required':True}),
    StructField("age", IntegerType(), True, {'age':'The user\'s age', \
                                             'required':False})])
# Other option
schema1 = StructType()
schema1.add(StructField("userId", IntegerType(), False))
schema1.add(StructField("name", StringType(), False, {'name':'The user\'s fullname', \
                                              'required':True}))
schema1.add(StructField("age", IntegerType(), True, {'age':'The user\'s age', \
                                            'required':False}))
```

1B. DEFINING A SCHEMA USING SPARK'S DDL

In Python, the types that can be used in a schema definition are describe here: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types.

Here are some examples:

```
schema2 = ["userId","name","age"]
schema3 = "userId int, name string, age int"
schema4 = ["userId int","name string","age int"]
```

11

2. CREATING A DATAFRAME FROM A SCHEMA

Use a SparkSession (spark) to create a DataFrame

```
objects1 = [[1, 'Fabio', 47], [2, 'Andrea', 47], [3, 'Thiago', 21]]
objects2 = [{'userId': 1, 'name': 'Fabio', 'age': 47}]

usersDf = spark.createDataFrame(objects1, schema1)
usersDf.printSchema()
usersDf.show()

aloneDf = spark.createDataFrame(objects2, schema2)
aloneDf.printSchema()
aloneDf.show()
```

ⓘ You can get the schema back using dataframe.schema (e.g. schema = usersDf.schema)

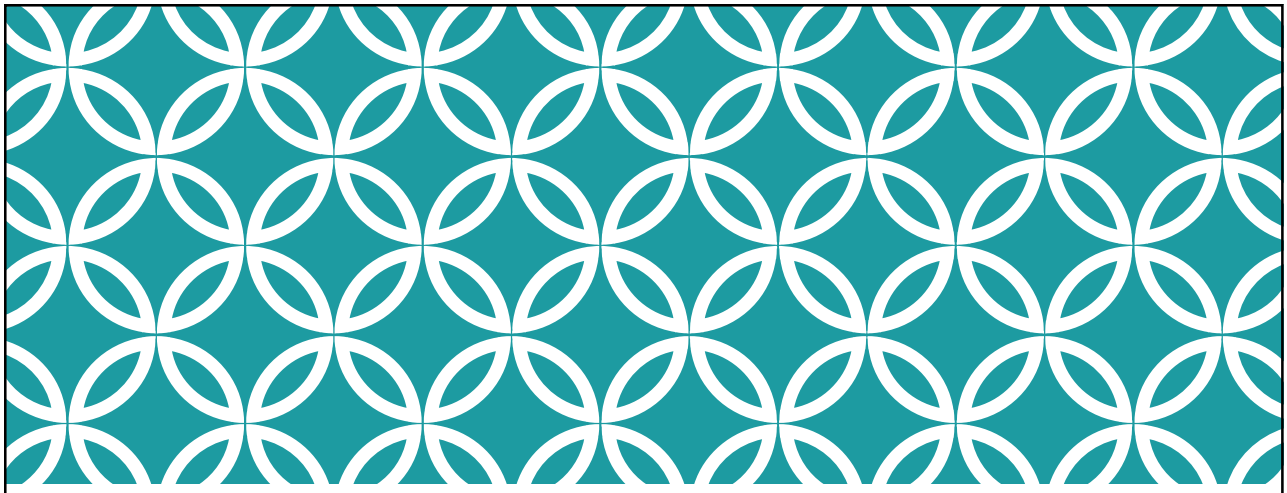
12

CREATING A DATAFRAME FROM AN EXTERNAL DATASOURCE

A SparkSession has a DataFrameReader that can be used to read data in as a DataFrame

```
usersCsvDf = spark.read.schema(schema1).csv('user.txt', )
# or spark.read.format('csv').option('sep','|').schema(usersFileSchema).load('u.user')
usersCsvDf.printSchema()
usersCsvDf.show()
```

SparkSession doc: https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.SparkSession



DATAFRAME MANIPULATION

14

CREATING A DATAFRAME FROM ANOTHER ONE

DataFrame class has methods representing typical relational operations, such as (e.g. select and where)

These be used to create a DataFrame from an existing one

15

SELECT AND SELECTEXP

Create a DataFrame from an existing one by selecting a subset of its columns

Recife a list of column names and/or expressions representing computations over the DataFrame's columns

```
# users1 has columns userId, name and age
from pyspark.sql.functions import col
namesAndAges = users1.select("name","age")
names = users1.select(users1["name"])
names = users1.select(users1.name)
names = users1.select(col("name"))
```

[2]:
```
from pyspark.sql import Row
from pyspark.sql.functions import col, expr

User = Row('id','firstName','lastName', 'age', 'salary')
bob = User(1,'Uncle','Bob', 58, 15000)
sally = User(2,'Sally','Fischer', 22, 2000)
miller = User(3,'Andersen','Miller', 44, 8000)

users = spark.createDataFrame([bob,sally,miller])
accountability = users.select( \
            'id', \
            col('salary').alias('dolars'), \
            expr('salary * 4').alias('reais'), \
            (col('salary') * 70).alias('pesos') \
           )
accountability.show()
```

```
+---+------+-----+-------+
| id|dolars|reais|  pesos|
+---+------+-----+-------+
|  1| 15000|60000|1050000|
|  2|  2000| 8000| 140000|
|  3|  8000|32000| 560000|
+---+------+-----+-------+
```

17

SELECTEXPR

Using the selectExpr method, we can specify valid SQL as strings

```
[3]: users.selectExpr('id', \
                      'salary as dolars', \
                      'salary * 4 as reais', \
                      'salary * 70 as pesos').show()
```

```
+---+------+-----+-------+
| id|dolars|reais|  pesos|
+---+------+-----+-------+
|  1| 15000|60000|1050000|
|  2|  2000| 8000| 140000|
|  3|  8000|32000| 560000|
+---+------+-----+-------+
```

```
[4]: accountability.selectExpr('count(*) as totalUsers', \
         'sum(dolars) as amount', \
         'avg(dolars) as salAvg').show()
```

```
+----------+------+-----------------+
|totalUsers|amount|           salAvg|
+----------+------+-----------------+
|         3| 25000|8333.333333333334|
+----------+------+-----------------+
```

LITERALS

To introduce a column that has a literal value, we shall use the lit function

```
[6]: from pyspark.sql.functions import lit

     employees = users.select('*', lit(2000).alias('baseSalary'))
     employees.show()
```

```
+---+---------+--------+---+------+----------+
| id|firstName|lastName|age|salary|baseSalary|
+---+---------+--------+---+------+----------+
|  1|    Uncle|     Bob| 58| 15000|      2000|
|  2|    Sally| Fischer| 22|  2000|      2000|
|  3| Andersen|  Miller| 44|  8000|      2000|
+---+---------+--------+---+------+----------+
```

ADDING, RENAMING, AND REMOVING COLUMNS

To create a new colum, use *withColumn* method

```
[7]: employees = users.withColumn('nextYearSalary',expr('salary * 1.1'))
     employees.show()
```

```
+---+---------+--------+---+------+--------------+
| id|firstName|lastName|age|salary|nextYearSalary|
+---+---------+--------+---+------+--------------+
|  1|    Uncle|     Bob| 58| 15000|       16500.0|
|  2|    Sally| Fischer| 22|  2000|        2200.0|
|  3| Andersen|  Miller| 44|  8000|        8800.0|
+---+---------+--------+---+------+--------------+
```

ADDING, RENAMING, AND REMOVING COLUMNS

To create a new DataFrame with a column renamed, use *withColumnRenamed* method

```
[8]: employees = employees.withColumnRenamed('nextYearSalary','projectedSalary')
     employees.show()
```

```
+---+---------+--------+---+------+---------------+
| id|firstName|lastName|age|salary|projectedSalary|
+---+---------+--------+---+------+---------------+
|  1|    Uncle|     Bob| 58| 15000|        16500.0|
|  2|    Sally| Fischer| 22|  2000|         2200.0|
|  3| Andersen|  Miller| 44|  8000|         8800.0|
+---+---------+--------+---+------+---------------+
```

ADDING, RENAMING, AND REMOVING COLUMNS

To create a new DataFrame without a column, use *drop* method

```
[10]: df = employees.drop('projectedSalary')
      df.show()
```

```
+---+---------+--------+---+------+
| id|firstName|lastName|age|salary|
+---+---------+--------+---+------+
|  1|    Uncle|     Bob| 58| 15000|
|  2|    Sally| Fischer| 22|  2000|
|  3| Andersen|  Miller| 44|  8000|
+---+---------+--------+---+------+
```


22

CHANGING COLUMN TYPES

Sometimes, we need to cast a column type… use *cast* method

```
[11]: # suppose that in users, salary is long
      salaries = users.select(col('salary').cast('decimal'))
      salaries.printSchema()
```

```
root
 |-- salary: decimal(10,0) (nullable = true)
```

FILTERING

Two methods equivalent methods available: *where* and *filter*

They receive a boolean expression, and select the rows to which it evaluates to *True*

To combine filters with a logical *and*, just chain them. To introduce a logical *or,* use the '|' operator.

```
[13]: lowSalaries = employees.where(col('salary') <= 3000)
      lowSalaries.show(5)
      mediumSalaries = employees.where(col('salary') > 3000).where(col('salary') < 10000)
      mediumSalaries.show(5)
      highSalaries = employees.where((expr('salary') >= 10000) | (col('projectedSalary') > 10000))
      highSalaries.show(5)
```

```
+---+---------+--------+---+------+---------------+
| id|firstName|lastName|age|salary|projectedSalary|
+---+---------+--------+---+------+---------------+
|  2|    Sally| Fischer| 22|  2000|         2200.0|
+---+---------+--------+---+------+---------------+

+---+---------+--------+---+------+---------------+
| id|firstName|lastName|age|salary|projectedSalary|
+---+---------+--------+---+------+---------------+
|  3| Andersen|  Miller| 44|  8000|         8800.0|
+---+---------+--------+---+------+---------------+

+---+---------+--------+---+------+---------------+
| id|firstName|lastName|age|salary|projectedSalary|
+---+---------+--------+---+------+---------------+
|  1|    Uncle|     Bob| 58| 15000|        16500.0|
+---+---------+--------+---+------+---------------+
```

COMPARE CONSIDERING NULLS

If there are null values in a column used in a filter, we shall use the *eqNullSafe* method

```
[15]: highSalaries.where(col("firstName").eqNullSafe("Uncle")).show()
```

```
+---+---------+--------+---+------+---------------+
| id|firstName|lastName|age|salary|projectedSalary|
+---+---------+--------+---+------+---------------+
|  1|    Uncle|     Bob| 58| 15000|        16500.0|
+---+---------+--------+---+------+---------------+
```

SELECT DISTINCT ROWS

```
[17]: users.distinct().show()
+---+---------+--------+---+------+
| id|firstName|lastName|age|salary|
+---+---------+--------+---+------+
|  3| Andersen|  Miller| 44|  8000|
|  2|    Sally| Fischer| 22|  2000|
|  1|    Uncle|     Bob| 58| 15000|
+---+---------+--------+---+------+
```


26

SORTING

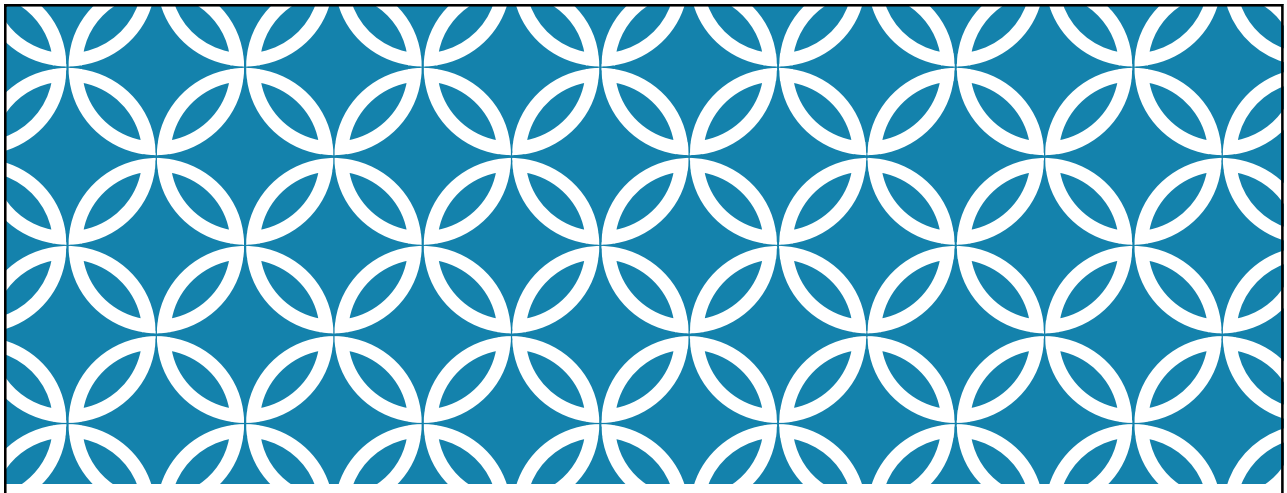
[19]:
```
User = Row('id','firstName','lastName', 'age', 'salary')
bob = User(1,'Uncle','Bob', 58, 15000)
sally = User(2,'Sally','Fischer', 22, 2000)
miller = User(3,'Andersen','Miller', 44, 8000)
bond = User(4, 'James', 'Bond', 40, 9999)
mike = User (5, 'Mike','Forster',24,2000)

users = spark.createDataFrame([bob,sally,miller, bond, mike])
employees = users.select('*', lit(2000).alias('baseSalary')).orderBy(col('salary').desc(),col('firstName').asc())
employees.show()

top3Sals = employees.limit(3)
top3Sals.show()
```

```
+---+---------+--------+---+------+----------+
| id|firstName|lastName|age|salary|baseSalary|
+---+---------+--------+---+------+----------+
|  1|    Uncle|     Bob| 58| 15000|      2000|
|  4|    James|    Bond| 40|  9999|      2000|
|  3| Andersen|  Miller| 44|  8000|      2000|
|  5|     Mike| Forster| 24|  2000|      2000|
|  2|    Sally| Fischer| 22|  2000|      2000|
+---+---------+--------+---+------+----------+

+---+---------+--------+---+------+----------+
| id|firstName|lastName|age|salary|baseSalary|
+---+---------+--------+---+------+----------+
|  1|    Uncle|     Bob| 58| 15000|      2000|
|  4|    James|    Bond| 40|  9999|      2000|
|  3| Andersen|  Miller| 44|  8000|      2000|
+---+---------+--------+---+------+----------+
```



QUESTIONS???

28