INTRODUCTION TO SPARK WITH SCALA

Spark SQL – Nested Data, UDF, Functions

Spark SQL

- Working with Columns
- Working with Nested Data
- User Defined Function (UDF)
- Having fun with Spark SQL functions (over 100+)
  - Date, String, Math, Misc.

Working with Columns

- Adding columns
- Renaming columns
- Dropping columns and rows
- Replace values in a column
- Filling the values of a column

Working with Columns

- Adding columns
  - Add data to an existing DataFrame
  - Combine columns into another column
- DataFrame.withColumn
  - Add or replacing existing column with same name

```
val data = Seq((1, "John Doe",  50),
               (2, "Mary Jane", 11))

val df = data.toDF("id", "name", "age")

val df1 = df.withColumn("isAdult", 'age > 15)
```

Working with Columns

- Renaming columns
  - User friendly column name
  - Column name collision during self-join
- DataFrame.withColumnRenamed
  - Rename an existing column
  - No-op if wrong column name is given

```
val data = Seq((1, "John Doe",  50),
               (2, "Mary Jane", 11))

val df = data.toDF("id", "name", "age")
val df1 = df.withColumnRenamed("name", "fullName")
```

Working with Columns

- Dropping columns
  - Data in the column is dirty, not useful
  - DataFrame.drop(colName)
- Dropping rows
  - DataFrame.na
  - DataFrameNaFunctions
    - drop()
      - drop rows contain null in any columns
    - drop(columnNames)
      - drop rows contain null in any of the given columns

Working with Columns

Dropping Column

```
val data = Seq((1, "John Doe",  50),
               (2, "Mary Jane", 11))

val df = data.toDF("id", "name", "age")
val df1 = df.drop(”id”)

df1.printSchema

root
  |-- name: string (nullable = true)
  |-- age: integer (nullable = false)
```

Working with Columns

Dropping Rows

```
val studentData = Seq(Row("Joe", 85), Row("Jane",
null))

val schema = StructType(Array(
   StructField("name", StringType, false),
   StructField("grade", IntegerType, true)
))

val rdd = spark.sparkContext.parallelize(studentData)

val studentDF = spark.createDataFrame( rdd, schema)

val newDF1 = studentDF.na.drop()
val newDF2 = studentDF.na.drop(Seq("name","grade"))
```

Working with Columns

- Replace/fill values in a column
  - Data in the column is dirty or missing
- org.apache.spark.sql.DataFrameNaFunctions
  - fill(value, cols)
    - fill the given value of specific type to the specified columns
  - replace(col, replaceMap)
  - replace(col2, replaceMap)
  - fill(value, columns)

Working with Columns


```
val studentData = Seq(Row("Joe", 85), Row("Jane", null))

val schema = StructType(Array(
   StructField("name", StringType, false),
   StructField("grade", IntegerType, true)
))

val rdd = spark.sparkContext.parallelize(studentData)

val studentDF = spark.createDataFrame( rdd, schema)

val newDF1 = studentDF.na.fill(0, Seq("grade"))
Val newDF2 = studentDF.na.replace("name", Map("Joe" ->
"John")).
```

Working with nested data

Company Products Data

```
|-- name: string
|-- year: long
|-- cities: array
|     |-- element: string
|-- products: array
|     |-- element: struct
|     |     |-- name: string
|     |     |-- year: long
```

Working with nested data


```
val json = """[
   {"name":"LinkedIn",
    "year":2002,
    "cities":["Mountain View", "Sunnyvale"],
    "products":[
         {"name":"Recruiter", "year":2006},
         {"name":"Slideshare", "year":2014}]
   },
   {"name":"Google",
    "year":1998,
    "cities":["Mountain View", "New York",],
    "products":[
      {"name":"GMail", "year":2004},
      {"name":"Android", "year":2008}]
   }
]"""
```

Working with nested data

- ***explode()*** function
  - Create one row per value in the array
  - Applicable for columns with array type

```
{″col1″, [“value1″, ″value2″] }
```

```
{″col1″, “value1″}
{″col1″, ″value2″}
```

Working with nested JSON

Working with nested JSON with explode function

```
val json = """<json string"""

import org.apache.spark.sql.functions._

val companies  = sqlContext.read.json(sc.parallelize(jsonString::Nil))

companies.select("name").show
companies.select("cities").show
companies.select("products").show

val cities = companies.select($"name",explode($"cities").as("city"))
cities.show

val products = companies.select($"name",
                   explode($"products").as("product"))
products.select($"product.name", $"product.year").show
```

User Defined Function (UDF)

- Custom data transformation
  - Functions that operate on data, row by row
- Can be written in Scala, Java, Python
  - Execute on worker machines
  - Performance issue with Python
- No code generation
  - Unlike built-in functions
  - Leverage built-in function first if available
- Steps
  - Write a Scala function
  - Register as UDF
  - Use registered UDF as a function

User Defined Function (UDF)


```
import org.apache.spark.sql.functions._

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

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


def letterGrade(score:Integer) : 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"
   }
}
```

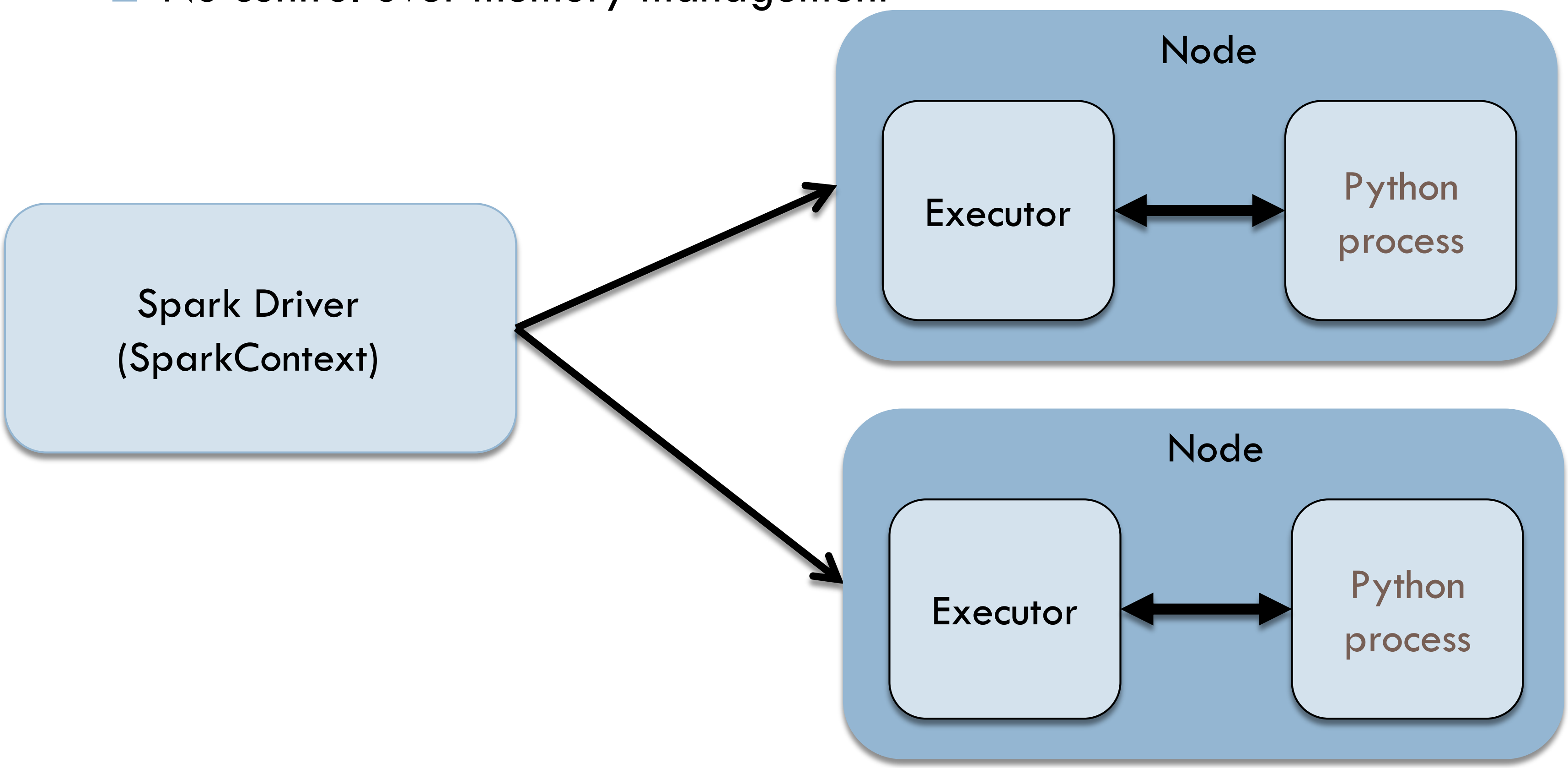
User Defined Function (UDF)

UDF Registration

```
import org.apache.spark.sql.functions.udf

// use as a DataFrame function
val letterGradeUDF = udf(letterGrade(_:Int))

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

// use in SQL statement
spark.sqlContext.udf.register("letterGrade",
                              letterGrade(_: Int): String)

studentDF.createOrReplaceTempView("students")

spark.sql("select name, letterGrade(score) from students")
```

User Defined Function (UDF)

- Performance overhead with Python UDF
  - Executors are JVM processes
  - Data serialization cost across processes
  - No control over memory management



Spark SQL Functions

- SQL function categories
  - Scalar
    - Single value for each row based on one more columns
  - Aggregation
    - Single value for a group of rows
  - Windows
    - Multiple values for a group of rows
  - User-defined
    - Scalar or aggregation

Spark SQL Functions

- Scala functions
  - Collections
  - Date/time
  - Math
  - Sorting
  - String
  - Miscellaneous

```
import org.apache.spark.sql.functions._
```

Spark SQL Functions

Category	Functions
Collection	from_json, to_json, size, sort_array
Date time	unix_timestamp, to_date, quarter, day, (da|week)yofyear, from_utc_timestap, to_utc_timestamp, year, month, dayofmonth, dayofweek, hour, minute, second, datediff, date_add, date_sub, add_months, last_day, next_day, months_between, current_date, current_time, trunc, date_format
Math	abs, bin, ceil, exp, floor, hex, log2, pow, round, toDegrees, toRadians
String	base64, concat, decode, encode, length, levenshtein, lower, trim, upper, regexp_replace,
Window	lag, lead, rank, percentRank
Misc	greatest, least, md5, sha, sha1, negate, when monotonicallyIncreasingId, sparkPartitionId

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

Spark SQL Functions


```
val data = List( (1, "2017-07-31", "2017-07-31 15:04:58.865"),
                 (2, "2017-06-11", "2017-07-15 17:24:18.326"),
                 (3, "2015-02-19", "2016-05-19 02:10:45.561"))

val dateDF = spark.sparkContext.parallelize(data)
                .toDF("id","joinDate", "txDate")

dateDF.select($"id",
              to_date($"joinDate").as("joinDate"),
              quarter($"joinDate").as("quarter"),
              dayofyear($"joinDate").as("doy"),
              weekofyear($"joinDate").as("woy"),
              last_day($"joinDate").as("lastday_month"),
              $"txDate",
              datediff(current_timestamp(),
              $"txDate").as("num_day_ago")).show
```

Spark SQL Functions


```
val data = List( (1, "2017-07-31", "2017-07-31 15:04:58.865"),
                 (2, "2017-06-11", "2017-07-15 17:24:18.326"),
                 (3, "2015-02-19", "2016-05-19 02:10:45.561"))
```

Output

```
+---+----------+-------+---+---+-------------+--------------------+-----------+
| id|  joinDate|quarter|doy|woy|lastday_month|              txDate|num_day_ago|
+---+----------+-------+---+---+-------------+--------------------+-----------+
|  1|2017-07-31|      3|212| 31|   2017-07-31|2017-07-31 15:04:...|          0|
|  2|2017-06-11|      2|162| 23|   2017-06-30|2017-07-15 17:24:...|         16|
|  3|2015-02-19|      1| 50|  8|   2015-02-28|2016-05-19 02:10:...|        438|
+---+----------+-------+---+---+-------------+--------------------+-----------+
```

Spark SQL Functions

- collect_list(col), collect_set(col)
  - Collect all values of a particular column after groupBy

```
val movies = List((“Titantic”,”Kate Winslet", 1997 ),
                  (“Titantic”,”DiCaprio, Leonardo", 1997),
                  (“Titantic”,”Cass, Marc", 1997))

val moviesDF = spark.sparkContext.parallelize(movies)
                .toDF(”title",”actor", ”year")

dateDF.groupBy(“title”)
     .agg(collect_list(“actor”).as(“actors”)).show
```

Spark SQL Functions

- Pivoting
  - Summarizing data by pivoting on categorical columns
  - Perform aggregations on other columns
  - The categorical values will become columns
- Use case
  - Given a list of students with their weight and graduation year
  - What is the average weight of students in each gender of each graduation year?

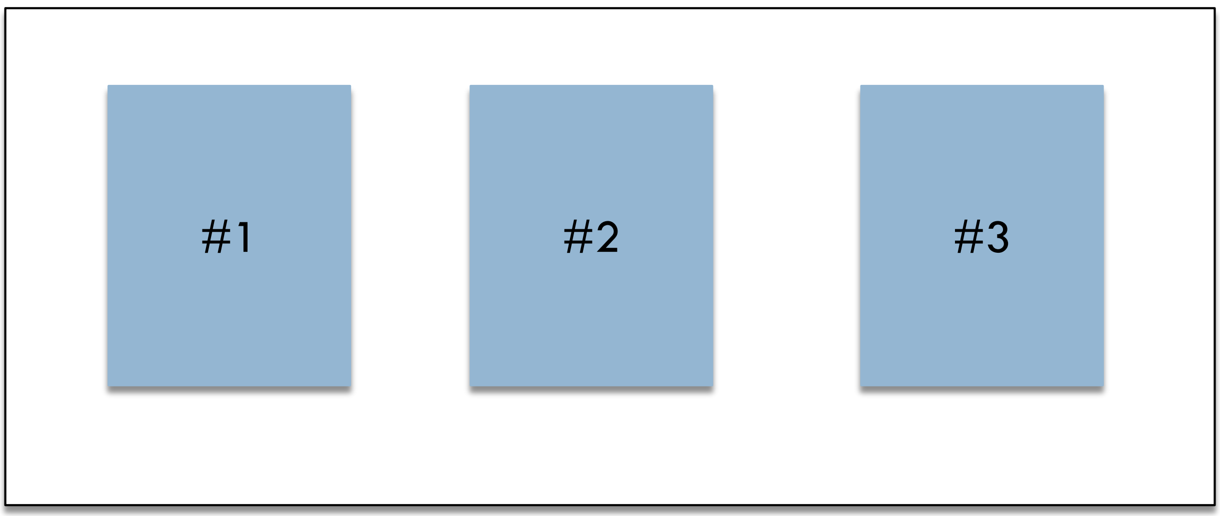
Spark SQL Functions


```
import org.apache.spark.sql.Row

case class Student(name:String, gender:String, weight:Int, grad_year:Int)

val studentsDF = Seq(Student("John", "M", 180, 2015),
                     Student("Mary", "F", 110, 2015),
                     Student("Derek", "M", 200, 2015),
                     Student("Julie", "F", 109, 2015),
                     Student("Allison", "F", 105, 2015),
                     Student("kirby", "F", 115, 2016),
                     Student("Jeff", "M", 195, 2016)).toDF

studentsDF.groupBy("graduation_year").pivot("gender").avg("weight").show()

+---------+-----+-----+
|grad_year|    F|    M|
+---------+-----+-----+
|     2015|108.0|190.0|
|     2016|115.0|195.0|
+---------+-----+-----+
```

Spark SQL Functions

- monotonically_increasing_id
  - Generating increasing ids
  - How to do this across many partitions?

DataFrame



Spark SQL Functions


```
val numDF = spark.range(1,11,1,5)

numDF.rdd.getNumPartitions

numDF.select('id, monotonically_increasing_id().as("m_ii"),
spark_partition_id().as("partition")).show

+---+-----------+---------+
| id|       m_ii|partition|
+---+-----------+---------+
|  1|          0|        0|
|  2|          1|        0|
|  3| 8589934592|        1|
|  4| 8589934593|        1|
|  5|17179869184|        2|
|  6|17179869185|        2|
|  7|25769803776|        3|
|  8|25769803777|        3|
|  9|34359738368|        4|
| 10|34359738369|        4|
+---+-----------+---------+
```

Spark SQL Functions

- Rollups & Cube
  - Good with hierarchical data, i.e sales
  - To generate reports with subtotal and grand total
  - Can rollup on more than 1 column

Spark SQL Functions


```
val q3Sales = List(("IPhone",2013, 35.2),
                   ("IPhone",2014, 47.53),
                   ("IPhone",2015, 40.40),
                   ("IPad",2013, 13.28),
                   ("IPad",2014, 10.93),
                   ("IPad",2015, 9.95),
                   ("Mac",2013, 4.41),
                   ("Mac",2014, 4.80),
                   ("Mac",2015, 4.25))

val q3SalesDF = spark.sparkContext.parallelize(q3Sales)
                     .toDF("product","year", "quantity")


q3SalesDF.rollup('product, 'year)
         .agg(sum("quantity") as "total")
         .orderBy('product.asc_nulls_last,
                  'year.asc_nulls_last).show
```

Spark SQL Functions


```
+-------+----+------+
|product|year| total|
+-------+----+------+
|   IPad|2013| 13.28|
|   IPad|2014| 10.93|
|   IPad|2015|  9.95|
|   IPad|null| 34.16|
| IPhone|2013|  35.2|
| IPhone|2014| 47.53|
| IPhone|2015|  40.4|
| IPhone|null|123.13|
|    Mac|2013|  4.41|
|    Mac|2014|   4.8|
|    Mac|2015|  4.25|
|    Mac|null| 13.46|
|   null|null|170.75|
+-------+----+------+
```

Spark SQL Functions - Window

- Window function
  - Compute a value for each row in a group of rows (frame)
  - A window specification determines what rows to include
  - Three different kind of window functions
    - Ranking, analytic and aggregate
  - Example: moving average or cumulative sums

Built-in functions return one value per row

Aggregation functions return one value per group of rows

Spark SQL Functions - Window

Customer Transaction Data

Name	Date	Amount
John	2017-07-02	15.35
John	2017-07-04	27.72
John	2017-07-06	21.33
Mary	2017-07-01	59.44
Mary	2017-07-03	99.76
Mary	2017-07-05	80.18
Mary	2017-07-07	69.74

1. What are top 2 spending amounts per user?
2. How does the spending amount trend over time?
3. Moving average, cumulative sum

Spark SQL Functions - Window

- Two step process
  - Define window specification
    - Partitioning – which rows will be in same partition
    - Ordering – how rows with each partition should be ordered
    - Frame – which rows will be included in the frame for the current row computation
  - Apply window function
    - rank, ntile, lag, lead

Spark SQL Functions - Window

Top 2 transactions per user

```
val custSpentData = List(
    ("John", "2017-07-02", 15.35), ("John", "2016-07-04", 27.72),
    ("John", "2016-07-06", 21.33), ("Mary", "2017-07-01", 59.44),
    ("Mary", "2017-07-03", 99.76), ("Mary", "2017-07-05", 80.18),
    ("Mary", "2017-07-07", 69.74))

val custSpentDataDF = spark.sparkContext.parallelize(custSpentData)
                           .toDF("name", "date", "amount")

val custSpentWindowSpec = Window.partitionBy("name").orderBy($”amount”.desc)

custSpentDataDF.withColumn("topSpent",
                  dense_rank().over(custSpentWindowSpec)).
                  where($"topSpent" <= 2).show
```

```
+----+----------+------+--------+
|name|      date|amount|topSpent|
+----+----------+------+--------+
|Mary|2017-07-03| 99.76|       1|
|Mary|2017-07-05| 80.18|       2|
|John|2016-07-04| 27.72|       1|
|John|2016-07-06| 21.33|       2|
+----+----------+------+--------+
```

Spark SQL Functions - Window

Difference Between Max Transaction Amount & Others

```
val spentWindowSpec = Window.partitionBy("name")
                            .rangeBetween(Window.unboundedPreceding,
                                          Window.unboundedFollowing)


val amountDiff = max(txDataDF("amount"))
                  .over(spentWindowSpec) - txDataDF("amount")

val txDiffWithHighestDF = custSpentDataDF.withColumn("amount_diff",
                                           round(amountDiff, 3))
```

Spark SQL Functions - Window


```
Amount Difference:

+----+----------+------+-----------+
|name|   tx_date|amount|amount_diff|
+----+----------+------+-----------+
|Mary|2017-07-05| 80.14|        0.0|
|Mary|2017-07-07| 69.74|       10.4|
|Mary|2017-07-01| 59.44|       20.7|
|John|2017-07-06| 27.33|        0.0|
|John|2017-07-04| 21.72|       5.61|
|John|2017-07-02| 13.35|      13.98|
+----+----------+------+-----------+
```

Spark SQL Functions - Window

Moving Average

```
val forMovingAvgWindow = Window.partitionBy("name")
                              .orderBy("tx_date")
                              .rowsBetween(Window.currentRow-1,
                                           Window.currentRow+1)

val txMovingAvgDF = txDataDF.withColumn("moving_avg",
                                         round(avg("amount")
                                .over(forMovingAvgWindow), 2))

txMovingAvgDF.show
```

Spark SQL Functions - Window


```
Moving average:

+----+----------+------+----------+
|name|   tx_date|amount|moving_avg|
+----+----------+------+----------+
|Mary|2017-07-01| 59.44|     69.79|
|Mary|2017-07-05| 80.14|     69.77|
|Mary|2017-07-07| 69.74|     74.94|
|John|2017-07-02| 13.35|     17.54|
|John|2017-07-04| 21.72|      20.8|
|John|2017-07-06| 27.33|     24.53|
+----+----------+------+----------+
```

Spark SQL Functions - Window

Accumulated Sum

```
val forCulmSumWindow = Window.partitionBy("name").orderBy("tx_date")
                              .rowsBetween(Window.unboundedPreceding,
                                           Window.currentRow)

val txCulmulativeSumDF = txDataDF.withColumn("culm_sum",
                                             round(sum("amount")
                                  .over(forCulmSumWindow),2))

txCulmulativeSumDF.show
```

Spark SQL Functions - Window


```
Culmulative sum:

+----+----------+------+--------+
|name|   tx_date|amount|culm_sum|
+----+----------+------+--------+
|Mary|2017-07-01| 59.44|   59.44|
|Mary|2017-07-05| 80.14|  139.58|
|Mary|2017-07-07| 69.74|  209.32|
|John|2017-07-02| 13.35|   13.35|
|John|2017-07-04| 21.72|   35.07|
|John|2017-07-06| 27.33|    62.4|
+----+----------+------+--------+
```