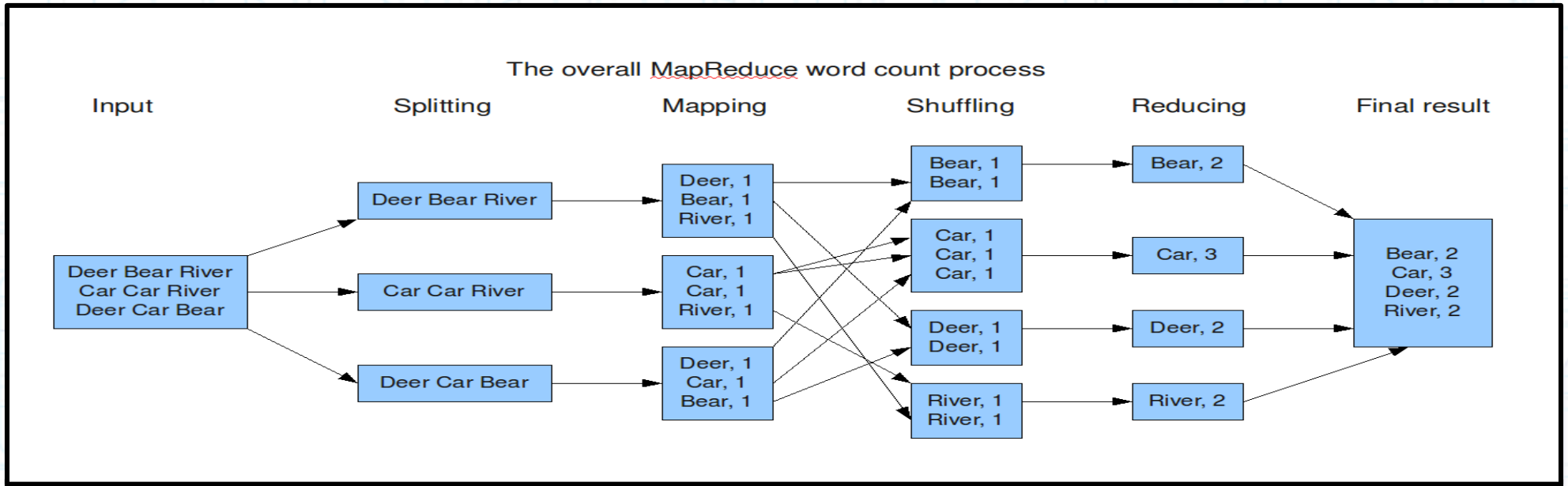
Especialização em Desenvolvimento de Software *Full Stack*

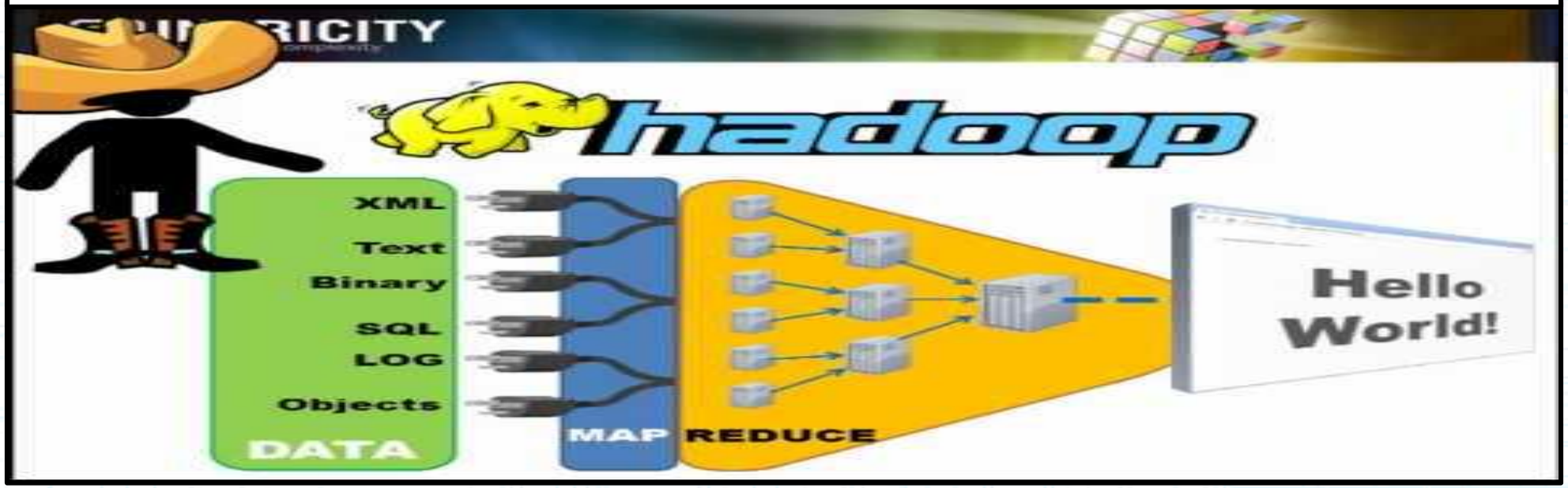
Módulo : - BIG DATA
Aula 02 – Levantando o Spark no Cluster Yarn e SparkSQL

Prof. Me. Luiz Mário Lustosa Pascoal

luizmlpascoal@gmail.com

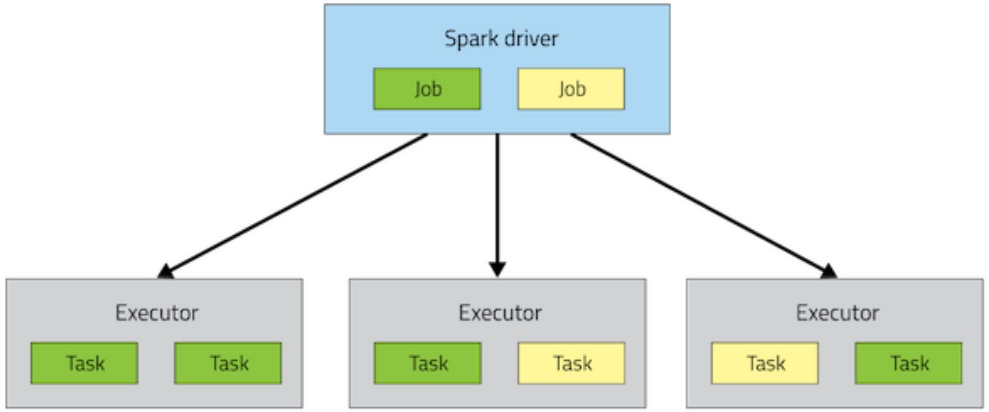
Funcionamento do Map Reduce





Arquitetura Spark

- Spark contém um coordenador central que administra os Jobs a serem executados.

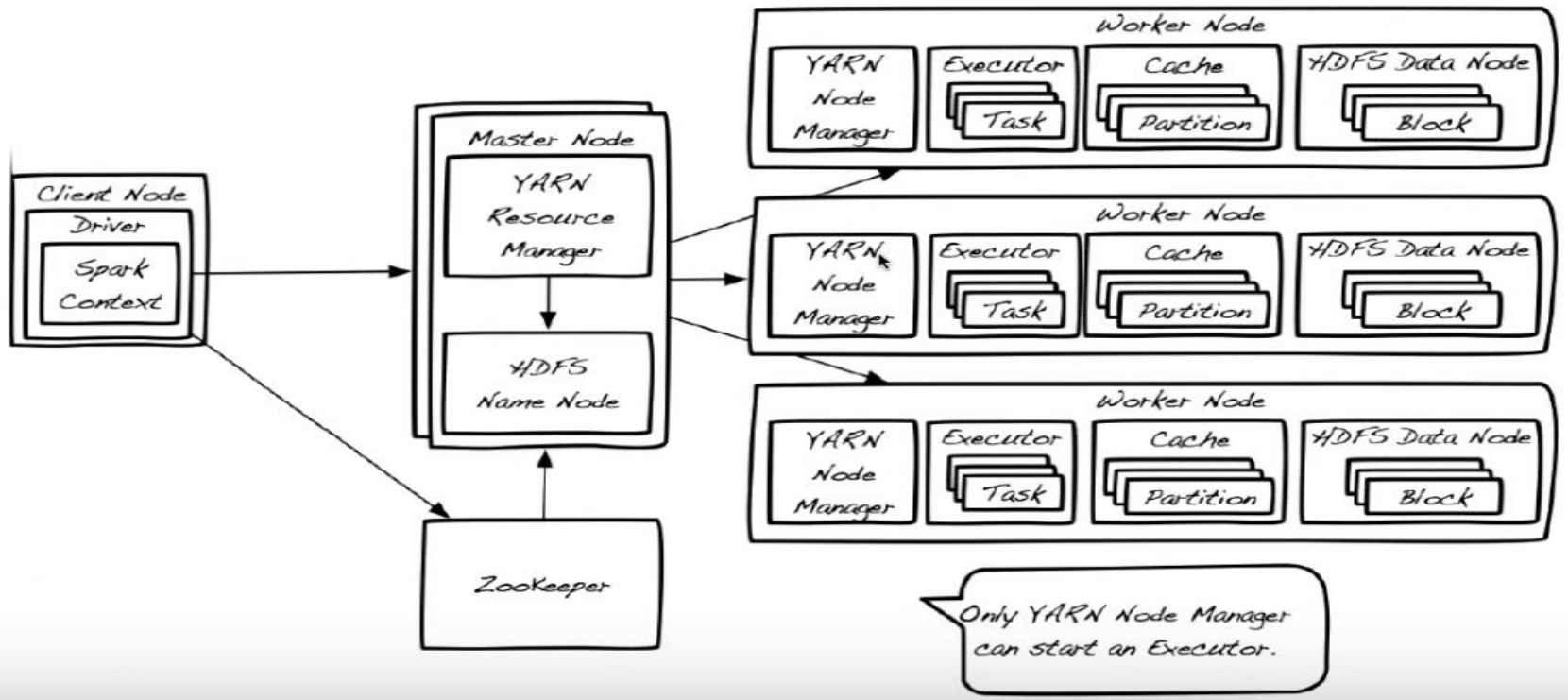


- Diferentemente do Hadoop MapReduce, cada aplicação terá processos (Executors) executando mesmo quando não se tem nenhum Job na fila.
- Isto permite armazenamento de dados em memória para acesso rápido e também inicialização de aplicação, logo maior VELOCIDADE.
- A desvantagem é que todo Executor tem capacidade de recursos fixa ao longo de toda execução da aplicação.

Cluster Management: Yarn

- O Cluster Manager é responsável por receber o Job a ser realizado e iniciar seu processamento nos Executors.
- O Spark tem suporte a três Cluster Managers:
  - StandAlone (Próprio do Spark)
    - Simples de usar
    - Executa os Jobs com organização FIFO
    - Toda aplicação usam todos os nós do cluster
  - Yarn (Yet Another Resource Negotiator)
    - Considerado como um Sistema Operacional para SDs
    - Suporta novas abordagens de processamento. Interligado com o HDFS.
    - Permite maior customização de recursos em diferentes Jobs
    - Jobs em Paralelo.
  - Mesos
    - Similar ao StandAlone
    - Executa em Kernels

YARN-based Architecture



- **Resource Manager:** Controle o uso de recursos dos Workers do cluster.
- **Node Manager:** Lança e monitora os jobs nos containers dos Worker Nodes do cluster.
- **Name Node:** Administra e controla as operações do HDFS.
- **Container:** Representa a coleção dos recursos físicos (CPU cores + memória) em cada nó do cluster. Estes recursos são utilizados pelos escravos (workers).

Por que usar o Yarn?

Executar o Spark no Yarn tem alguns benefícios:

- O YARN permite o compartilhamento dinâmico de recursos do cluster entre diferentes frameworks que rodam no YARN. Exemplo: Pode-se executar um Job MapReduce e logo em seguida executar um Spark Job sem nenhuma mudança nas configurações.
- Pode-se utilizar o Organizador de tarefas do YARN para categorizar, isolar e priorizar alguns workloads.
- YARN é o único cluster manager para o Spark que têm suporte a segurança, pois é baseado totalmente autenticação entre os processos e de acesso aos nós do cluster.

Vamos a configuração....

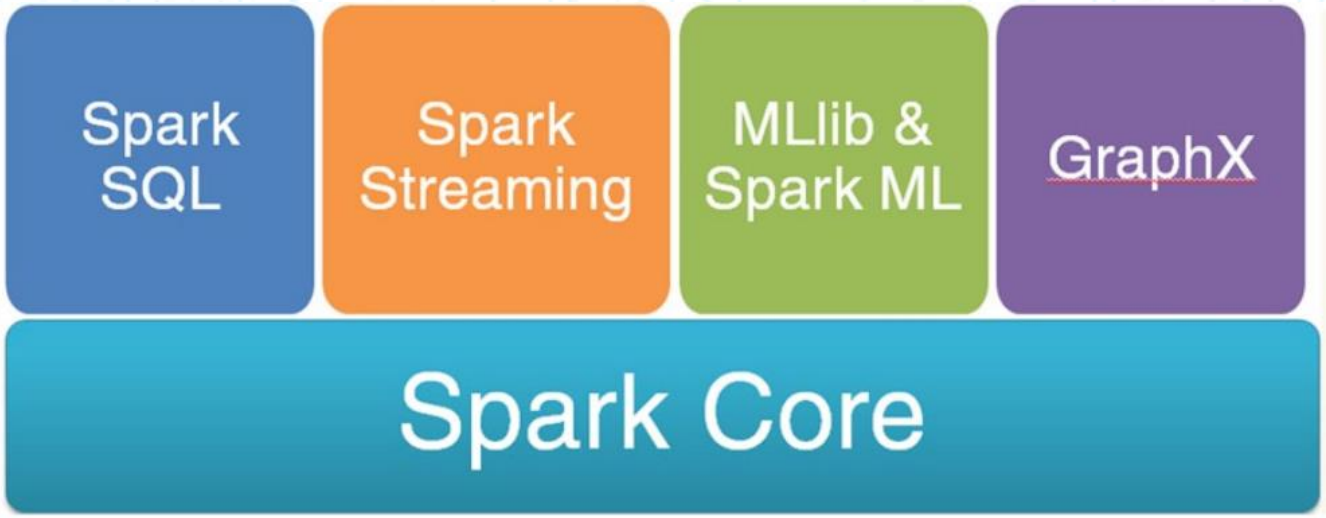
- Arquivo de texto “passo a passo.txt” para apoio.
- Sites para o Tutorial:

  (1) https://linode.com/docs/databases/hadoop/how-to-install-and-set-up-hadoop-cluster/

  (2) https://linode.com/docs/databases/hadoop/install-configure-run-spark-on-top-of-hadoop-yarn-cluster/

Façam grupos de 3 pessoas.

SparkSQL



Arquitetura – SQL

- Baseado em Schema – Trabalha com dados estruturados
  - Databases relacionais
  - Arquivos CSV
  - Arquivos JSON
- Suporta múltiplas linguagens de consulta
  - SQL
  - HiveQL
  - Linguagem de consulta integrada

Arquitetura – SQL

- Integrado aos demais módulos do Spark
- Simplicidade de uso – Programação Declarativa
- Integração com diversas fontes de dados
  - PostgreSQL, MySQL, H2, Oracle, DB2, MS SQL Server, entre outros com JDBC
  - Hbase, Cassandra, Elasticsearch, Druid, entre outros noSQL

Arquitetura – SQL

- Preocupação forte com Performance
  - Redução de *I/O* de disco
  - Armazenamento em colunas ao invés de linhas
  - Particionamento dos dados horizontal
  - Trabalha com os dados binários na memória off-heap
  - Assim como em banco de dados tradicionais, é possível fazer otimizações da *query*

Arquitetura – SQL

- Abstrações de API
  - DataFrame
    - Row
  - Dataset
- Requer o uso do SQLContext

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

val config = new SparkConf().setAppName("My Spark SQL app")
val sc = new SparkContext(config)
val sqlContext = new SQLContext(sc)
```

Input/Output – SQL

- Input
  - A partir de um RDD

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

val config = new SparkConf().setAppName("My Spark SQL app")
val sc = new SparkContext(config)
val sqlContext = new SQLContext(sc)
import sqlContext.implicits._

case class Employee(name: String, age: Int, gender: String)

val rowsRDD = sc.textFile("path/to/employees.csv")
val employeesRDD = rowsRDD.map{row => row.split(",")}
                          .map{cols => Employee(cols(0), cols(1).trim.toInt, cols(2))}

val employeesDF = employeesRDD.toDF()
```

Input/Output – SQL

- Input
  - A partir de um RDD

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

val config = new SparkConf().setAppName("My Spark SQL app")
val sc = new SparkContext(config)
val sqlContext = new SQLContext(sc)

val linesRDD = sc.textFile("path/to/employees.csv")
val rowsRDD = linesRDD.map{row => row.split(",")}
                      .map{cols => Row(cols(0), cols(1).trim.toInt, cols(2))}
val schema = StructType(List(
                StructField("name", StringType, false),
                StructField("age", IntegerType, false),
                StructField("gender", StringType, false)
              )
            )

val employeesDF = sqlContext.createDataFrame(rowsRDD,schema)
```

Input/Output – SQL

- Input
  - A partir de arquivos (data sources)

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

val config = new SparkConf().setAppName("My Spark SQL app")
val sc = new SparkContext(config)
val sqlContext = new org.apache.spark.sql.hive.HiveContext (sc)

// create a DataFrame from parquet files
val parquetDF = sqlContext.read
                          .format("org.apache.spark.sql.parquet")
                          .load("path/to/Parquet-file-or-directory")

// create a DataFrame from JSON files
val jsonDF = sqlContext.read
                       .format("org.apache.spark.sql.json")
                       .load("path/to/JSON-file-or-directory")
```

Input/Output – SQL

- Input
  - A partir de arquivos (data sources)
    - JSON
    - Cada objeto JSON deve estar em 1 linha

```
val jsonHdfsDF = sqlContext.read.json("hdfs://NAME_NODE/path/to/data.json")
val jsonS3DF = sqlContext.read.json("s3a://BUCKET_NAME/FOLDER_NAME/data.json")
```

Input/Output – SQL

- Input
  - A partir de arquivos (data sources)
    - JSON
    - Cada objeto JSON deve estar em 1 linha

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

val userSchema = StructType(List(
                    StructField("name", StringType, false),
                    StructField("age", IntegerType, false),
                    StructField("gender", StringType, false)
                    )
                  )

val userDF = sqlContext.read
                       .schema(userSchema)
                       .json("path/to/user.json")
```

Input/Output – SQL

- Input
  - A partir de arquivos (data sources)
    - JDBC
    - val hiveContext = new HiveContext(sc)

```
val jdbcUrl ="jdbc:mysql://host:port/database"
val tableName = "table-name"
val connectionProperties = new java.util.Properties
connectionProperties.setProperty("user","database-user-name")
connectionProperties.setProperty("password"," database-user-password")

val jdbcDF = hiveContext.read
                        .jdbc(jdbcUrl, tableName, connectionProperties)
```

Input/Output – SQL

- Output
  - Método
    - write
    - configurações
    - save

```
// save a DataFrame in JSON format
customerDF.write
  .format("org.apache.spark.sql.json")
  .save("path/to/output-directory")
```

```
// save a DataFrame in Parquet format
homeDF.write
  .format("org.apache.spark.sql.parquet")
  .partitionBy("city")
  .save("path/to/output-directory")
```

```
// save a DataFrame as a Postgres database table
df.write
  .format("org.apache.spark.sql.jdbc")
  .options(Map(
     "url" -> "jdbc:postgresql://host:port/database?user=<USER>&password=<PASS>",
     "dbtable" -> "schema-name.table-name"))
   .save()
```

Transformações/Ações – SQL

- Operações básicas
  - **cache**
  - columns
  - dtypes
  - explain
  - persist
  - printSchema
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Cria um cache do Dataframe no formato de colunas. Apenas as mais usadas.

Melhora a performance e pode ser "*tunado*".

```
customerDF.cache()
```

```
sqlContext.setConf("spark.sql.inMemoryColumnarStorage.compressed", "true")
sqlContext.setConf("spark.sql.inMemoryColumnarStorage.batchSize", "10000")
```

Transformações/Ações – SQL

- Operações básicas
  - cache
  - **columns**
  - dtypes
  - explain
  - persist
  - printSchema
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Retorna um array de String contendo o nome das colunas.

```
val cols = customerDF.columns
```

```
cols: Array[String] = Array(cId, name, age, gender)
```

Transformações/Ações – SQL

- Operações básicas
  - cache
  - columns
  - **dtypes**
  - explain
  - persist
  - printSchema
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Semelhante ao Columns, porém retorn um vetor de tuplas, cada uma com o nome da coluna e seu tipo.

```
val columnsWithTypes = customerDF.dtypes
```

```
columnsWithTypes: Array[(String, String)] = Array((cId,LongType), (name,StringType),
(age,IntegerType), (gender,StringType))
```

Transformações/Ações – SQL

- Operações básicas
  - cache
  - columns
  - dtypes
  - **explain**
  - persist
  - printSchema
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Imprime no console a estrutura física do RDD. Pode ser útil para análise. Mas pouco usado durante o desenvolvimento.

```
customerDF.explain()
```

```
== Physical Plan ==
InMemoryColumnarTableScan [cId#0L,name#1,age#2,gender#3], (InMemoryRelation
[cId#0L,name#1,age#2,gender#3], true, 10000, StorageLevel(true, true, false, true, 1),
(Scan PhysicalRDD[cId#0L,name#1,age#2,gender#3]), None)
```

Transformações/Ações – SQL

- Operações básicas
  - cache
  - columns
  - dtypes
  - explain
  - **persist**
  - printSchema
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Faz um *"match point" e cria uma cache* em memória do Dataframe. Utíl para recuperações De falhas nos *workers*.

```
customerDF.persist
```

Transformações/Ações – SQL

- Operações básicas
  - cache
  - columns
  - dtypes
  - explain
  - persist
  - **printSchema**
  - registerTempTable
    - createOrReplaceTempView
  - toDF

Imprime no console o Schema do DataFrame no formato de árvore.

```
customerDF.printSchema()
```

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

Transformações/Ações – SQL

- Operações básicas
  - cache
  - columns
  - dtypes
  - explain
  - persist
  - printSchema
  - **registerTempTable**
    - **createOrReplaceTempView**
  - toDF

Cria uma tabela temporária no contexto do HIVE
Com os dados do DataFrame.
Útil para trabalhar com SQL

```
customerDF.registerTempTable("customer")
val countDF = sqlContext.sql("SELECT count(1) AS cnt FROM customer")
```

Transformações/Ações – SQL

- Operações básicas
  - Cache
  - Columns
  - Dtypes
  - Explain
  - Persist
  - PrintSchema
  - RegisterTempTable
    - createOrReplaceTempView
  - **toDF**

Permite renomear as colunas de um DataFrame.

```
val resultDF = sqlContext.sql("SELECT count(1) from customer")

resultDF: org.apache.spark.sql.DataFrame = [_c0: bigint]

val countDF = resultDF.toDF("cnt")

countDF: org.apache.spark.sql.DataFrame = [cnt: bigint]
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - **agg**
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Aplica funções de agregação no DataFrame.

```
val aggregates = productDF.agg(max("price"), min("price"), count("name"))
```

```
aggregates: org.apache.spark.sql.DataFrame = [max(price): double, min(price): double,
count(name): bigint]
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - **apply**
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - Limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Recupera uma determinada coluna do DataFrame.

```
val priceColumn = productDF.apply("price")
```

```
priceColumn: org.apache.spark.sql.Column = price
```

```
val discountedPriceColumn = priceColumn * 0.5
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - **cube**
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Recebe o nome de uma ou mais colunas e retorna um cubo mutidemensional para analise dessas colunas (somatório).

```
val salesCubeDF = salesDF.cube($"date", $"product", $"country").sum("revenue")
```

```
salesCubeDF: org.apache.spark.sql.DataFrame = [date: string, product: string, country: string, sum(revenue): double]
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - **distinct**
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Cria um nova DataFrame com elementos distintos do original.

```
val dfWithoutDuplicates = customerDF.distinct
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - **explode**
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF com 0 ou mais linhas para cada linha do DF original, baseado em alguma Função do usuário.

```
val wordDF = emailDF.explode("body", "word") { body: String => body.split(" ")}
wordDF.show
```

```
+------+---------+-------+--------------------+--------+
|sender|recepient|subject|                body|    word|
+------+---------+-------+--------------------+--------+
| James|     Mary|   back|just got back fro...|    just|
| James|     Mary|   back|just got back fro...|     got|
| James|     Mary|   back|just got back fro...|    back|
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - **filter**
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF com apenas os elementos do DF original que satisfaçam a função de filtro.

```
val filteredDF = customerDF.filter("age > 25")
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - **groupBy**
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF aplicando alguma operação de Agregação agrupando por alguma coluna.

```
val countByGender = customerDF.groupBy("gender").count
```

```
val revenueByProductDF = salesDF.groupBy("product").sum("revenue")
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - **intersect**
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Aplicado a dois DFs. Gera um novo DF com Elementos que em comum aos dois Dfs originais.

```
val customers2 = List(Customer(11, "Jackson", 21, "M"),
                      Customer(12, "Emma", 25, "F"),
                      Customer(13, "Olivia", 31, "F"),
                      Customer(4, "Jennifer", 45, "F"),
                      Customer(5, "Robert", 41, "M"),
                      Customer(6, "Sandra", 45, "F"))

val customer2DF = sc.parallelize(customers2).toDF()
val commonCustomersDF = customerDF.intersect(customer2DF)
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - **join**
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF aplicando uma operação de Join entre dois DFs.

```
case class Transaction(tId: Long, custId: Long, prodId: Long, date: String, city: String)
val transactions = List(Transaction(1, 5, 3, "01/01/2015", "San Francisco"),
                        Transaction(2, 6, 1, "01/02/2015", "San Jose"),
                        Transaction(3, 1, 6, "01/01/2015", "Boston"),
                        Transaction(4, 200, 400, "01/02/2015", "Palo Alto"),
                        Transaction(6, 100, 100, "01/02/2015", "Mountain View"))

val transactionDF = sc.parallelize(transactions).toDF()
val innerDF = transactionDF.join(customerDF, $"custId" === $"cId", "inner")
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - Join
  - **limit**
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF contendo uma quantidade limitada dos elementos do DF original.

```
val fiveCustomerDF = customerDF.limit(5)
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - **orderBy**
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera um novo DF com os dados do DF original
Porem ordenados segundo um conjunto de coleções.

```
val sortedDF = customerDF.orderBy("name")
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - **randomSplit**
  - rollup
  - sample
  - select
  - selectExpr
  - withColumn

Gera novos DFs de forma randomica a partir de um DF original. Muito útil para ML.

```
val dfArray = homeDF.randomSplit(Array(0.6, 0.2, 0.2))
dfArray(0).count
dfArray(1).count
dfArray(2).count
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - **rollup**
  - sample
  - select
  - selectExpr
  - withColumn

Similar ao cube. Porém executa uma operação de rollup. Útil para subagregações.

```
case class SalesByCity(year: Int, city: String, state: String,
                       country: String, revenue: Double)
val salesByCity = List(SalesByCity(2014, "Boston", "MA", "USA", 2000),
                       SalesByCity(2015, "Boston", "MA", "USA", 3000),
                       SalesByCity(2014, "Cambridge", "MA", "USA", 2000),
                       SalesByCity(2015, "Cambridge", "MA", "USA", 3000),
                       SalesByCity(2014, "Palo Alto", "CA", "USA", 4000),
                       SalesByCity(2015, "Palo Alto", "CA", "USA", 6000),
                       SalesByCity(2014, "Pune", "MH", "India", 1000),
                       SalesByCity(2015, "Pune", "MH", "India", 1000),
                       SalesByCity(2015, "Mumbai", "MH", "India", 1000),
                       SalesByCity(2014, "Mumbai", "MH", "India", 2000))
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - **rollup**
  - sample
  - select
  - selectExpr
  - withColumn

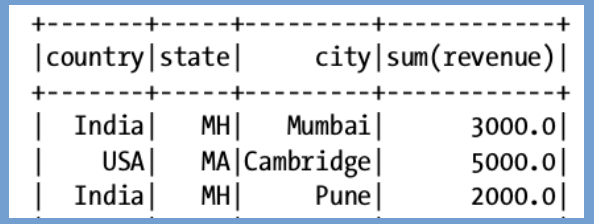
Similar ao cube. Porém executa uma operação de rollup. Útil para subagregações.

```
val salesByCityDF = sc.parallelize(salesByCity).toDF()
val rollup = salesByCityDF.rollup($"country", $"state", $"city").sum("revenue")
rollup.show
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - **rollup**
  - sample
  - select
  - selectExpr
  - withColumn

Similar ao cube. Porém executa uma operação de rollup. Útil para subagregações.



```
+-------+-----+---------+-----------+
|country|state|     city|sum(revenue)|
+-------+-----+---------+-----------+
|  India|   MH|   Mumbai|     3000.0|
|    USA|   MA|Cambridge|     5000.0|
|  India|   MH|     Pune|     2000.0|
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - **sample**
  - select
  - selectExpr
  - withColumn

Gera um novo DF com uma fração dos elementos do DF original.

```
val sampleDF = homeDF.sample(true, 0.10)
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - **select**
  - selectExpr
  - withColumn

Gera um novo DF sendo uma projeção do DF original, usando as colunas selecionadas.

```
val namesAgeDF = customerDF.select("name", "age")
```

```
val newAgeDF = customerDF.select($"name", $"age" + 10)
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - **selectExpr**
  - withColumn

Similar ao select, porém aceita expressões como argumentos.

```
val newCustomerDF = customerDF.selectExpr("name", "age + 10  AS new_age",
                                          "IF(gender = 'M', true, false) AS male")
```

Transformações/Ações – SQL

- Queries da linguagem integrada
  - agg
  - apply
  - cube
  - distinct
  - explode
  - filter
  - groupBy
  - intersect
  - join
  - limit
  - orderBy
  - randomSplit
  - rollup
  - sample
  - select
  - selectExpr
  - **withColumn**

Cria um novo DF, adicionando ou alterando uma coluna do DF original.

```
val newProductDF = productDF.withColumn("profit", $"price" - $"cost")
```

Transformações/Ações – SQL

- Operações de RDD
  - rdd
  - toJson

Transformações/Ações – SQL

- Operações de RDD
  - **rdd**
  - toJson

Converte o DataFrame para um RDD de Row
Contendo cada elemento como Row.

```
import org.apache.spark.sql.Row
val rdd = customerDF.rdd
```

```
rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[113] at rdd at
<console>:28
```

Transformações/Ações – SQL

- Operações de RDD
  - rdd
  - **toJson**

Converte o DataFrame para um RDD de String Contendo cada elemento como String no formato JSON.

```
val jsonRDD = customerDF.toJSON
```

```
jsonRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[408] at toJSON at
<console>:28
```

Transformações/Ações – SQL

- Ações
  - collect
  - count
  - describe
  - first
  - show
  - take

Transformações/Ações – SQL

- Ações
  - **collect**
  - count
  - describe
  - first
  - show
  - take

Retorna os elementos que estão distribuídos no cluster para o Driver Program, como um array de vetores.

```
val result = customerDF.collect
```

```
result: Array[org.apache.spark.sql.Row] = Array([1,James,21,M], [2,Liz,25,F],
[3,John,31,M], [4,Jennifer,45,F], [5,Robert,41,M], [6,Sandra,45,F])
```

Transformações/Ações – SQL

- Ações
  - collect
  - **count**
  - describe
  - first
  - show
  - take

Retorna a quantidade de linhas no DataFrame

```
val count = customerDF.count
```

```
count: Long = 6
```

Transformações/Ações – SQL

- Ações
  - collect
  - count
  - **describe**
  - first
  - show
  - take

Descreva o tipo dos dados do DataFrame.

```
val summaryStatsDF = productDF.describe("price", "cost")
```

```
summaryStatsDF: org.apache.spark.sql.DataFrame = [summary: string, price: string, cost: string]
```

Transformações/Ações – SQL

- Ações
  - collect
  - count
  - describe
  - **first**
  - show
  - take

Retorna para o Driver Program a primeira Row do DataFrame

```
val first = customerDF.first

first: org.apache.spark.sql.Row = [1,James,21,M]
```

Transformações/Ações – SQL

- Ações
  - collect
  - count
  - describe
  - first
  - **show**
  - take

Apresenta os dados do DataFrame no console formatados como uma tabela.

```
summaryStatsDF.show

+-------+------------------+------------------+
|summary|             price|              cost|
+-------+------------------+------------------+
|  count|                 6|                 6|
|   mean| 566.6666666666666| 416.6666666666667|
| stddev|309.12061651652357|240.94720491334928|
|    min|             200.0|             100.0|
|    max|            1200.0|             900.0|
+-------+------------------+------------------+
```

Transformações/Ações – SQL

- Ações
  - collect
  - count
  - describe
  - first
  - show
  - **take**

Retorna para o Driver Program '*n*' elementos do Dataframe no formato de um array.

```
val first2Rows = customerDF.take(2)
```

```
first2Rows: Array[org.apache.spark.sql.Row] = Array([1,James,21,M], [2,Liz,25,F])
```

Transformações/Ações – SQL

- Funções internas
  - Agregações
  - Date/Time
  - Matemática
  - String

Transformações/Ações – SQL

- Funções internas
  - **Agregações**
  - Date/Time
  - Matemática
  - String

Executa agregações sobre os dados:

avg, count, countDistinct, first, last, max, mean, min, sum, sumDistinct, etc

Transformações/Ações – SQL

- Funções internas
  - Agregações
  - **Date/Time**
  - Matemática
  - String

Funções para trabalhar com datas, por exemplo:

unix_timestamp, from_unixtime,date_add, date_sub, add_months, last_day, next_day

Transformações/Ações – SQL

- Funções internas
  - Agregações
  - Date/Time
  - **Matemática**
  - String

Executa operações matemáticas sobre os itens.
Como por exemplo:

abs, ceil, cos, exp, factorial, floor, hex, hypot, log, log10, pow, round, shiftLeft, sin, sqrt, tan.

Transformações/Ações – SQL

- Funções internas
  - Agregações
  - Date/Time
  - Matemática
  - **String**

Faz transformações sobre elementos do tipo String.

ascii, base64, concat,split, substring, substring_index, translate, trim, etc.

Hands-on – SQL

Análise Interativa com SparkSQL

Hands-on – SQL

- Dataset
  - Portal da transparência → Diárias de viagem → 2017 → janeiro
  - http://portaltransparencia.gov.br/downloads/mensal.asp?c=Diarias#meses01

iconv -f ISO-8859-1 -t UTF-8 input.csv > output.csv

Hands-on – SQL

- Qual ministério mais viajou?
- Qual ministério mais gastou?

- Exemplo de criação do DF

```
//Cria um DataFrame usando o spark Session (funciona desde o Spark 2.0+)
val df = spark.read.option("header", true).option("inferSchema",
true).csv("/home/etl/exemplosTeste/Ministerio/diariasConv.csv")

//Usando separador de colunas com espaço "vários options"
val df = spark.read.option("header", true).option("inferSchema", true).option("sep",
"\t").csv("/home/etl/exemplosTeste/Ministerio/diariasConv.csv")
```

Exercício para Entrega/Avaliação

Entrar no Portal de Transparência (http://portaltransparencia.gov.br/downloads/)

Escolha dois tipos de dados (Despesas, Pagamentos, Programas Sociais, Receitas etc.) e realize operações de *data mining* nestes dados com cruzamentos e análises sobre como o dinheiro público têm sido investido.

Entregar um documento descritivo do dataset utilizado e da operação realizada, bem como os comandos implementados para realizar a tarefa via mensagem pessoal para o professor no Slack.

Referências

Guller, Mohammed. **Big Data Analytics with Spark**. Apress, 2015.

-
- Duvvuri, Srinivas, and Bikramaditya Singhal. **Spark for Data Science**. Packt Publishing Ltd, 2016.
-
- KARAU, Holden et al. **Learning spark: lightning-fast big data analysis**. " O'Reilly Media, Inc.", 2015.
-
- DEROOS, Dirk et al. **Hadoop for Dummies**. John Wiley & Sons, Incorporated, 2014.