Spark Programming – Spark SQL

Bu eğitim sunumları İstanbul Kalkınma Ajansı'nın 2016 yılı Yenilikçi ve Yaratıcı İstanbul Mali Destek Programı kapsamında yürütülmekte olan TR10/16/YNY/0036 no'lu İstanbul Big Data Eğitim ve Araştırma Merkezi Projesi dahilinde gerçekleştirilmiştir. İçerik ile ilgili tek sorumluluk Bahçeşehir Üniversitesi'ne ait olup İSTKA veya Kalkınma Bakanlığı'nın görüşlerini yansıtmamaktadır.

Spark SQL

blurs the lines between RDDs and relational tables

intermix SQL commands to query external data, along with complex analytics, in a single app:

- allows SQL extensions based on MLlib
- Shark is being migrated to Spark SQL

Spark SQL

```scala
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/
people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")


// The results of SQL queries are SchemaRDDs
// normal RDD operations.
// The columns of a row in the result can be
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

Hive Interoperability

- Spark SQL is compatible with Hive.
- It not only supports HiveQL, but can also access Hive metastore, SerDes, and UDFs.
- You can also replace Hive with Spark SQL to get better performance.
- HiveQL queries run much faster on Spark SQL than on Hive.

Spark SQL: queries in HiveQL

```
//val sc: SparkContext // An existing SparkContext.
//NB: example on laptop lacks a Hive MetaStore
val hiveContext = new
org.apache.spark.sql.hive.HiveContext(sc)

// Importing the SQL context gives access to all the
// public SQL functions and implicit conversions.
import hiveContext._
hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
hql("FROM src SELECT key,
value").collect().foreach(println)
```

Executing SQL Queries Programmatically

The SQLContext class provides a method named **sql**, which executes a SQL query using Spark.

It takes a SQL statement as an argument and returns the result as an instance of the **DataFrame** class.

```
val resultSet = sqlContext.sql("SELECT count(1) FROM my_table")
```

DataFrame

DataFrame is Spark SQL's primary data abstraction.

- Unlike RDD, DataFrame is schema aware.
- It represents a distributed collection of rows organized into named columns. Conceptually, it is similar to a table in a relational database.

DataFrame Row

Row is a Spark SQL abstraction for representing a row of data.

- Conceptually, it is equivalent to a relational tuple or row in a table.
- Spark SQL provides factory methods to create Row objects. An example is shown next.

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

val row1 = Row("Barack Obama", "President", "United States")
val row2 = Row("David Cameron", "Prime Minister", "United Kingdom")
```

The value of a column in a Row object can be fetched using its ordinal. Examples are shown next.

```
val presidentName = row1.getString(0)
val country = row1.getString(2)
```

Creating a DataFrame

A DataFrame can be created in two ways.

- it can be created from a data source.
- a DataFrame can be created from an RDD.

Spark SQL provides two methods for creating a DataFrame from an RDD: **toDF** and **createDataFrame**.

Creating a DataFrame using toDF

Spark SQL provides an implicit conversion method named toDF, which creates a DataFrame from an RDD of objects represented by a case class.

- Spark SQL infers the schema of a dataset.
- The toDF method is not defined in the RDD class, but it is available through an implicit conversion.
- To convert an RDD to a DataFrame using toDF, you need to import the implicit methods defined in the implicits object.

```
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()
```

createDataFrame

```
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)
```

The createDataFrame method takes two arguments, an RDD of Rows and a schema, and returns a DataFrame. abstraction

- The schema for a dataset can be specified with an instance of StructType, which is a case class.
- A StructType object contains a sequence of StructField objects.
- StructField is also defined as a case class.
- The key difference between the toDF and createDataFrame methods is that the former infers the schema of a dataset and the latter requires you to specify the schema.

Creating a DataFrame from a Data Source

```
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")

// create a DataFrame from a table in a Postgres database
val jdbcDF = sqlContext.read
               .format("org.apache.spark.sql.jdbc")
               .options(Map(
                 "url" -> "jdbc:postgresql://host:port/database?user=<USER>&password=<PASS>",
                 "dbtable" -> "schema-name.table-name"))
             .load()

// create a DataFrame from a Hive table
val hiveDF = sqlContext.read
                       .table("hive-table-name")
```

Spark SQL provides a unified interface for creating a DataFrame from a variety of data sources.

- Spark SQL provides a class named ***DataFrameReader***, which defines the interface for reading data froma data source.
- It allows you to specify different options for reading data

For example, the same API can be used to create a DataFrame from a MySQL, PostgreSQL, Oracle, or Cassandra table.

DataFrame from JSON using schema

The *DataFrameReader* class provides a method named json for reading a JSON dataset.

- It takes a path as argument and returns a DataFrame.
- The path can be the name of either a JSON file or a directory containing multiple JSON files.

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

- Spark SQL automatically infers the schema of a JSON dataset by scanning the entire dataset to determine the schema.
- Can avoid scan and speed up DataFrame creation by specifying schema.

```
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")
```

Processing Data Programmatically with SQL/HiveQL

The **sql** method in the **HiveContext** class allows using HiveQL, whereas the **sql** method in the **SQLContext** class allows using SQL statements.

- The table referenced in a SQL/HiveQL statement must have an entry in a Hive metastore.
- If not, can create a temporary table using the registerTempTable method provided by the DataFrame class.
- The sql method returns result as a DataFrame, for displaying the returned result on a console or saving it to a data source.

```
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")
```

```
userDF.registerTempTable("user")
val cntDF = hiveContext.sql("SELECT count(1) from user")
val cntByGenderDF = hiveContext.sql(
                    "SELECT gender, count(1) as cnt FROM user GROUP BY gender ORDER BY cnt")
```

Processing Data with the DataFrame API

The DataFrame API provides an alternative way for processing a dataset.

```
$ cd SPARK_HOME
$ ./bin/spark-shell --master local[*]
```

Once inside the Spark-shell, create a few DataFrames.

```
case class Customer(cId: Long, name: String, age: Int, gender: String)
val customers = List(Customer(1, "James", 21, "M"),
                     Customer(2, "Liz", 25, "F"),
                     Customer(3, "John", 31, "M"),
                     Customer(4, "Jennifer", 45, "F"),
                     Customer(5, "Robert", 41, "M"),
                     Customer(6, "Sandra", 45, "F"))

val customerDF = sc.parallelize(customers).toDF()

case class Product(pId: Long, name: String, price: Double, cost: Double)
val products = List(Product(1, "iPhone", 600, 400),
                    Product(2, "Galaxy", 500, 400),
                    Product(3, "iPad", 400, 300),
                    Product(4, "Kindle", 200, 100),
                    Product(5, "MacBook", 1200, 900),
                    Product(6, "Dell", 500, 400))

val productDF = sc.parallelize(products).toDF()

case class Home(city: String, size: Int, lotSize: Int,
                bedrooms: Int, bathrooms: Int, price: Int)
val homes = List(Home("San Francisco", 1500, 4000, 3, 2, 1500000),
                 Home("Palo Alto", 1800, 3000, 4, 2, 1800000),
                 Home("Mountain View", 2000, 4000, 4, 2, 1500000),
                 Home("Sunnyvale", 2400, 5000, 4, 3, 1600000),
                 Home("San Jose", 3000, 6000, 4, 3, 1400000),
                 Home("Fremont", 3000, 7000, 4, 3, 1500000),
                 Home("Pleasanton", 3300, 8000, 4, 3, 1400000),
                 Home("Berkeley", 1400, 3000, 3, 3, 1100000),
                 Home("Oakland", 2200, 6000, 4, 3, 1100000),
                 Home("Emeryville", 2500, 5000, 4, 3, 1200000))

val homeDF = sc.parallelize(homes).toDF
```

Basic DataFrame Operations: cache

The cache method stores the source DataFrame in memory using a columnar format.

- It scans only the required columns and stores them in compressed in-memory columnar format.
- Spark SQL automatically selects a compression codec for each column based on data statistics.

```
customerDF.cache()
```

The caching functionality can be tuned using the setConf method in the SQLContext or HiveContext class.
The two configuration parameters for caching are

- spark.sql.inMemoryColumnarStorage.compressed
- and spark.sql.inMemoryColumnarStorage.batchSize.

By default, compression is turned on and the batch size for columnar caching is 10,000.

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

DataFrame columns and dtypes

The columns method returns the names of all the columns in the source DataFrame as an array of String.

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

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

The dtypes method returns the data types of all the columns in the source DataFrame as an array of tuples.
The first element in a tuple is the name of a column and the second element is the data type of that column.

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

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

explain, printSchema methods

The explain method prints the physical plan on the console. It is useful for debugging.

```
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)
```

The printSchema method prints the schema of the source DataFrame on the console in a tree format

```
customerDF.printSchema()
```

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

registerTempTable, toDF methods

The registerTempTable method creates a temporary table in Hive metastore.

- It takes a table name as an argument and sql method returns a DataFrame.
- A temporary table can be queried using the sql method in SQLContext or HiveContext.
- It is available only during the lifespan of the application that creates it.

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

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

The toDF method allows you to rename the columns in the source DataFrame. It takes new names of the columns as arguments and returns new 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]
```

Language-Integrated Query Methods: agg

The **agg** is a commonly used language-integrated query methods of the DataFrame class. This method performs specified aggregations on one or more columns in the source DataFrame and returns the result as a new 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]
```

```
aggregates.show
```

```
+----------+----------+-----------+
|max(price)|min(price)|count(name)|
+----------+----------+-----------+
|    1200.0|     200.0|          6|
+----------+----------+-----------+
```

Language-Integrated Query Methods: apply

The apply method takes the name of a column as an argument and returns the specified column in the source DataFrame as an instance of the Column class.

- The Column class provides operators for manipulating a column in a DataFrame.

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

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

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

```
discountedPriceColumn: org.apache.spark.sql.Column = (price * 0.5)
```

Scala allows using productDF("price") instead of productDF.apply("price")

- It automatically converts productDF("price") to productDF.apply("price")

```
val priceColumn = productDF("price")
val discountedPriceColumn = priceColumn * 0.5
```

distinct

If a method or function expects an instance of the Column class as an argument, you can use the $"... " notation to select a column in a DataFrame.

The following three statements are equivalent.

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

The **distinct** method returns a new DataFrame containing only the unique rows in the source DataFrame.

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

cube

The cube method returns a cube for multi-dimensional analysis.

- It is useful for generating cross-tabular reports.
- Assume you have a dataset that tracks sales along three dimensions: time, product and country.
- The cube method generates aggregates for all the possible combinations of the dimensions.

```
case class SalesSummary(date: String, product: String, country: String, revenue: Double)
val sales = List(SalesSummary("01/01/2015", "iPhone", "USA", 40000),
                 SalesSummary("01/02/2015", "iPhone", "USA", 30000),
                 SalesSummary("01/01/2015", "iPhone", "China", 10000),
                 SalesSummary("01/02/2015", "iPhone", "China", 5000),
                 SalesSummary("01/01/2015", "S6", "USA", 20000),
                 SalesSummary("01/02/2015", "S6", "USA", 10000),
                 SalesSummary("01/01/2015", "S6", "China", 9000),
                 SalesSummary("01/02/2015", "S6", "China", 6000))

val salesDF = sc.parallelize(sales).toDF()

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

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

```
salesCubeDF.withColumnRenamed("sum(revenue)", "total").show(30)
```

```
+----------+-------+-------+--------+
|      date|product|country|   total|
+----------+-------+-------+--------+
|01/01/2015|   null|    USA| 60000.0|
|01/02/2015|     S6|   null| 16000.0|
|01/01/2015| iPhone|   null| 50000.0|
|01/01/2015|     S6|  China|  9000.0|
|      null|   null|  China| 30000.0|
|01/02/2015|     S6|    USA| 10000.0|
|01/02/2015|   null|   null| 51000.0|
|01/02/2015| iPhone|  China|  5000.0|
```

If you wanted to find the total sales of all products in the USA, you can use the following expression.

```
salesCubeDF.filter("product IS null AND date IS null AND country='USA'").show
```

```
+----+-------+-------+------------+
|date|product|country|sum(revenue)|
+----+-------+-------+------------+
|null|   null|    USA|    100000.0|
+----+-------+-------+------------+
```

explode

The explode method generates zero or more rows from a column using a user-provided function.
It takes three arguments:

- input column,
- output column
- user provided function generating one or more values for the output column for each value in the input column.

For example, consider a text column containing contents of an email.

- to split the email content into individual words and a row for each word in an email.

```
case class Email(sender: String, recepient: String, subject: String, body: String)
val emails = List(Email("James", "Mary", "back", "just got back from vacation"),
                  Email("John", "Jessica", "money", "make million dollars"),
                  Email("Tim", "Kevin", "report", "send me sales report ASAP"))

val emailDF = sc.parallelize(emails).toDF()
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|
| James|     Mary|   back|just got back fro...|    from|
| James|     Mary|   back|just got back fro...|vacation|
|  John|  Jessica|  money|make million dollars|    make|
|  John|  Jessica|  money|make million dollars| million|
|  John|  Jessica|  money|make million dollars| dollars|
|   Tim|    Kevin| report|send me sales rep...|    send|
|   Tim|    Kevin| report|send me sales rep...|      me|
|   Tim|    Kevin| report|send me sales rep...|   sales|
|   Tim|    Kevin| report|send me sales rep...|  report|
|   Tim|    Kevin| report|send me sales rep...|    ASAP|
+------+---------+-------+------------------+--------+
```

filter

The filter method filters rows in the source DataFrame using a SQL expression provided to it as an argument.
It returns a new DataFrame containing only the filtered rows.
The SQL expression can be passed as a string argument.

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

```
filteredDF: org.apache.spark.sql.DataFrame = [cId: bigint, name: string, age: int, gender: string]
```

```
filteredDF.show
```

```
+---+--------+---+------+
|cId|    name|age|gender|
+---+--------+---+------+
|  3|    John| 31|     M|
|  4|Jennifer| 45|     F|
|  5|  Robert| 41|     M|
|  6|  Sandra| 45|     F|
+---+--------+---+------+
```

A variant of the filter method allows a filter condition to be specified using the Column type.

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

As mentioned earlier, the preceding code is a short-hand for the following code.

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

groupBy

The groupBy method groups the rows in the source DataFrame using the columns provided to it as arguments.
Aggregation can be performed on the grouped data returned by this method.

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

```
countByGender: org.apache.spark.sql.DataFrame = [gender: string, count: bigint]
```

```
countByGender.show
```

```
+------+-----+
|gender|count|
+------+-----+
|     F|    3|
|     M|    3|
+------+-----+
```

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

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

```
+-------+------------+
|product|sum(revenue)|
+-------+------------+
| iPhone|     85000.0|
|     S6|     45000.0|
+-------+------------+
```

intersect

The intersect method takes a DataFrame as an argument and returns a new DataFrame containing only the rows in both the input and source DataFrame

```
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)
commonCustomersDF.show
```

```
+---+--------+---+------+
|cId|    name|age|gender|
+---+--------+---+------+
|  6|  Sandra| 45|     F|
|  4|Jennifer| 45|     F|
|  5|  Robert| 41|     M|
+---+--------+---+------+
```

join

The join method performs a SQL join of the source DataFrame with another DataFrame.

It takes three arguments, a DataFrame, a join expression and a join type.

```
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")
```

```
innerDF.show
```

```
+---+------+------+----------+-------------+---+------+---+------+
|tId|custId|prodId|      date|         city|cId|  name|age|gender|
+---+------+------+----------+-------------+---+------+---+------+
|  1|     5|     3|01/01/2015|San Francisco|  5|Robert| 41|     M|
|  2|     6|     1|01/02/2015|     San Jose|  6|Sandra| 45|     F|
|  3|     1|     6|01/01/2015|       Boston|  1| James| 21|     M|
+---+------+------+----------+-------------+---+------+---+------+
```

```
val leftOuterDF = transactionDF.join(customerDF, $"custId" === $"cId", "left_outer")
leftOuterDF.show
```

```
+---+------+------+----------+-------------+----+------+----+------+
|tId|custId|prodId|      date|         city| cId|  name| age|gender|
+---+------+------+----------+-------------+----+------+----+------+
|  1|     5|     3|01/01/2015|San Francisco|   5|Robert|  41|     M|
|  2|     6|     1|01/02/2015|     San Jose|   6|Sandra|  45|     F|
|  3|     1|     6|01/01/2015|       Boston|   1| James|  21|     M|
|  4|   200|   400|01/02/2015|    Palo Alto|null|  null|null|  null|
|  6|   100|   100|01/02/2015|Mountain View|null|  null|null|  null|
+---+------+------+----------+-------------+----+------+----+------+
```

limit, orderBy

The limit method returns a DataFrame containing the specified number of rows from the source DataFrame

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

```
+---+--------+---+------+
|cId|    name|age|gender|
+---+--------+---+------+
|  1|   James| 21|     M|
|  2|     Liz| 25|     F|
|  3|    John| 31|     M|
|  4|Jennifer| 45|     F|
|  5|  Robert| 41|     M|
+---+--------+---+------+
```

The orderBy method returns a DataFrame sorted by the given columns. It takes the names of one or more columns as arguments.

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

```
+---+--------+---+------+
|cId|    name|age|gender|
+---+--------+---+------+
|  1|   James| 21|     M|
|  4|Jennifer| 45|     F|
|  3|    John| 31|     M|
|  2|     Liz| 25|     F|
|  5|  Robert| 41|     M|
|  6|  Sandra| 45|     F|
+---+--------+---+------+
```

```
val sortedByAgeNameDF = customerDF.sort($"age".desc, $"name".asc)
sortedByAgeNameDF.show
```

```
+---+--------+---+------+
|cId|    name|age|gender|
+---+--------+---+------+
|  4|Jennifer| 45|     F|
|  6|  Sandra| 45|     F|
|  5|  Robert| 41|     M|
|  3|    John| 31|     M|
|  2|     Liz| 25|     F|
|  1|   James| 21|     M|
+---+--------+---+------+
```

randomSplit, sample

The randomSplit method splits the source DataFrame into multiple DataFrames. It takes an array of weights as argument and returns an array of DataFrames. It is a useful method for machine learning, where you want to split the raw dataset into training, validation and test datasets.

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

The sample method returns a DataFrame containing the specified fraction of the rows in the source DataFrame.
It takes two arguments.

- The first argument is a Boolean value indicating whether sampling should be done with replacement.
- The second argument specifies the fraction of the rows that should be returned.

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

rollup

The rollup method takes the names of one or more columns as arguments and returns a multi-dimensional rollup.

It is useful for subaggregation along a hierarchical dimension such as geography or time.

```
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))

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

```
+-------+-----+---------+------------+
|country|state|     city|sum(revenue)|
+-------+-----+---------+------------+
|  India|   MH|   Mumbai|      3000.0|
|    USA|   MA|Cambridge|      5000.0|
|  India|   MH|     Pune|      2000.0|
|    USA|   MA|   Boston|      5000.0|
|    USA|   MA|     null|     10000.0|
|    USA| null|     null|     20000.0|
|    USA|   CA|     null|     10000.0|
|   null| null|     null|     25000.0|
|  India|   MH|     null|      5000.0|
|    USA|   CA|Palo Alto|     10000.0|
|  India| null|     null|      5000.0|
+-------+-----+---------+------------+
```

select

The select method returns a DataFrame containing only the specified columns from the source DataFrame.

A variant of the select method allows one or more Column expressions as arguments.

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

```
+--------+---+
|    name|age|
+--------+---+
|   James| 21|
|     Liz| 25|
|    John| 31|
|Jennifer| 45|
|  Robert| 41|
|  Sandra| 45|
+--------+---+
```

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

```
+--------+----------+
|    name|(age + 10)|
+--------+----------+
|   James|        31|
|     Liz|        35|
|    John|        41|
|Jennifer|        55|
|  Robert|        51|
|  Sandra|        55|
+--------+----------+
```

selectExpr

The selectExpr method accepts one or more SQL expressions as arguments

returns a DataFrame generated by executing the specified SQL expressions.

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

newCustomerDF.show
```

```
+--------+-------+-----+
|    name|new_age| male|
+--------+-------+-----+
|   James|     31| true|
|     Liz|     35|false|
|    John|     41| true|
|Jennifer|     55|false|
|  Robert|     51| true|
|  Sandra|     55|false|
+--------+-------+-----+
```

withColumn

The withColumn method adds a new column to or replaces an existing column in the source DataFrame and returns a new DataFrame.

It takes two arguments:

- the name of the new column
- an expression for generating the values of the new column.

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

```
+---+-------+------+-----+------+
|pId|   name| price| cost|profit|
+---+-------+------+-----+------+
|  1| iPhone| 600.0|400.0| 200.0|
|  2| Galaxy| 500.0|400.0| 100.0|
|  3|   iPad| 400.0|300.0| 100.0|
|  4| Kindle| 200.0|100.0| 100.0|
|  5|MacBook|1200.0|900.0| 300.0|
|  6|   Dell| 500.0|400.0| 100.0|
+---+-------+------+-----+------+
```

RDD Operations

The DataFrame class supports commonly used RDD operations such as map, flatMap, foreach, foreachPartition, mapPartition, coalesce, and repartition.

- These methods work similar to the operations in the RDD class.
- if you need access to other RDD methods that are not present in the DataFrame class, can get an RDD from a DataFrame.

```
val rdd = customerDF.rdd
```

```
rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[405] at rdd at
<console>:27
```

```
val firstRow = rdd.first
```

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

```
val name = firstRow.getString(1)
```

```
name: String = James
```

```
val age = firstRow.getInt(2)
```

```
age: Int = 21
```

RDD Operations

Fields in a Row can also be extracted using Scala pattern matching.

```
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
```

```
val nameAndAge = rdd.map {
                   case Row(cId: Long, name: String, age: Int, gender: String) => (name, age)
                 }
```

```
nameAndAge: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[114] at map at
<console>:30
```

```
nameAndAge.collect
```

```
res79: Array[(String, Int)] = Array((James,21), (Liz,25), (John,31), (Jennifer,45),
(Robert,41), (Sandra,45))
```

DataFrame Actions

Similar to the RDD actions, the action methods in the DataFrame class return results to the Driver program.

collect

The collect method returns the data in a DataFrame as an array of Rows.

```
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])
```

count

The count method returns the number of rows in the source DataFrame.

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

```
count: Long = 6
```

DataFrame Actions: describe

The describe method can be used for exploratory data analysis.

- It returns summary statistics for numeric columns in the source DataFrame.
- The summary statistics includes min, max, count, mean, and standard deviation.

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

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

```
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|
+-------+------------------+------------------+
```

DataFrame Actions: first, show, take

The **first** method returns the first row in the source DataFrame.

```
val first = customerDF.first
```

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

The **show** method displays the rows in the source DataFrame on the driver console in a tabular format.
Optionally displays the top N rows. By default, it shows the top 20.

```
customerDF.show(2)
```

```
+---+-----+---+------+
|cId| name|age|gender|
+---+-----+---+------+
|  1|James| 21|     M|
|  2|  Liz| 25|     F|
+---+-----+---+------+
```

The **take** method takes an integer N as an argument and returns the first N rows from the source DataFrame as an array of Rows.

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

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

Saving a DataFrame

Spark SQL provides a unified interface for saving a DataFrame to a variety of data sources

The same interface can be used to write data to relational databases, NoSQL data stores and a variety of file formats.

The DataFrameWriter class defines the interface for writing data to a data source.

```
// 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 in ORC file format
homeDF.write
  .format("orc")
  .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()

// save a DataFrame to a Hive table
df.write.saveAsTable("hive-table-name")
```

SparkSQL Built-in Functions

Spark SQL comes with a comprehensive list of built-in functions, which are optimized for fast execution.

- The built-in functions can be used from both the DataFrame API and SQL interface.
- To use Spark's built-in functions from the DataFrame API, you need to add the following import statement to your source code.

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

The built-in functions can be classified into the following categories:

- aggregate,
- collection,
- date/time,
- math,
- string,
- window, and
- miscellaneous functions.

Aggregate

The aggregate functions can be used to perform aggregations on a column.

The built-in aggregate functions include

- approxCountDistinct,
- avg,
- count,
- countDistinct,
- first,
- last,
- max,
- mean,
- min,
- sum, and
- sumDistinct.

```
val minPrice = homeDF.select(min($"price"))
minPrice.show
```

```
+----------+
|min(price)|
+----------+
|   1100000|
+----------+
```

Collection, Date/Time functions

The collection functions operate on columns containing a collection of elements.
The built-in collection functions include *array_contains, explode, size, and sort_array.*

The date/time functions make it easy to process columns containing date/time values.
These functions can be further sub-classified into the following categories: *conversion, extraction, arithmetic, and miscellaneous functions.*

Conversion, Field Extraction, Arithmetic

The **conversion functions** convert date/time values from one format to another. For example, you can convert a timestamp string in yyyy-MM-dd HH:mm:ss format to a Unix epoch value using the unix_timestamp function.

- The built-in conversion functions include unix_timestamp, from_unixtime, to_date, quarter, day, dayofyear, weekofyear, from_utc_timestamp, and to_utc_timestamp.

The **field extraction functions** allow you to extract year, month, day, hour, minute, and second from a Date/Time value.

- The built-in field extraction functions include year, quarter, month, weekofyear, dayofyear, dayofmonth, hour, minute, and second.

The **arithmetic functions** allow you to perform arithmetic operation on columns containing dates. For example, you can calculate the difference between two dates, add days to a date, or subtract days from a date.

- The built-in date arithmetic functions include datediff, date_add, date_sub, add_months, last_day, next_day, and months_between.

Miscellaneous functions

Spark SQL provides a few other useful date- and **time-related functions**:

- current_date, current_timestamp, trunc, date_format.

The **math functions** operate on columns containing numerical values. Spark SQL comes with a long list of built-in math functions.

- abs, ceil, cos, exp, factorial, floor, hex, hypot, log, log10, pow, round, shiftLeft, sin, sqrt, tan, and other commonly used math functions.

The **string functions**: Spark SQL provides a variety of built-in functions for processing columns that contain string values.

- The built-in string functions include ascii, base64,concat, concat_ws, decode, encode, format_number, format_string, get_json_object, initcap, instr,length, levenshtein, locate, lower, lpad, ltrim, printf, regexp_extract, regexp_replace, repeat,reverse, rpad, rtrim, soundex, space, split, substring, substring_index, translate, trim, unbase64, upper, and other commonly used string functions

Spark SQL supports **window functions** for analytics. A window function performs a calculation across a set of rows that are related to the current row.

- The built-in window functions provided by Spark SQL include cumeDist, denseRank, lag, lead, ntile, percentRank, rank, and rowNumber.

Interactive Analysis Example

launch the Spark shell from a terminal,
path/to/spark/bin/spark-shell --master local[*]

For using a few classes and functions from the Spark SQL library, use import statement.

import org.apache.spark.sql._

create a DataFrame from a dataset.
val biz = sqlContext.read.json("path/to/yelp_academic_dataset_business.json")

```
sqlContext.sql("SELECT count(1) as businesses FROM biz").show
```

```
+----------+
|businesses|
+----------+
|     61184|
+----------+
```

Language-Integrated Query vs SQL

```
biz.filter(biz("stars") <=> 5.0)
   .select("name","stars", "review_count", "city",  "state")
   .show(5)
```

The preceding code first uses the filter method to filter the businesses that have average rating of 5.0.

You could have also written the language integrated query version as follows:

```
sqlContext.sql("SELECT name, stars, review_count, city, state FROM biz WHERE stars=5.0").show(5)
```

```
+--------------------+-----+------------+------------+-----+
|                name|stars|review_count|        city|state|
+--------------------+-----+------------+------------+-----+
|    Alteration World|  5.0|           5|    Carnegie|   PA|
|American Buyers D...|  5.0|           3|   Homestead|   PA|
|Hunan Wok Chinese...|  5.0|           4|West Mifflin|   PA|
|      Minerva Bakery|  5.0|           7|  McKeesport|   PA|
|               Vivo|  5.0|           3|    Bellevue|   PA|
+--------------------+-----+------------+------------+-----+
```