/CursoAmadeus20...

```
%md
Datasets use with this tutorial
---
```

FINISHED

```
%sh
ls -lt /Users/juantomas/proyectos/cursos/curso-spark/datasets
```

FINISHED

```
total 1347920
-rw-r--r--@ 1 juantomas  staff  689413344 28 feb 22:07 2008.csv
-rw-r--r--@ 1 juantomas  staff     244438 28 feb 22:07 airports.csv
-rw-r--r--@ 1 juantomas  staff      43758 28 feb 22:07 carriers.csv
-rw-r--r--@ 1 juantomas  staff     428796 28 feb 22:07 plane-data.csv
```

Took 0 sec. Last updated by anonymous at February 28 2017, 10:13:28 PM.

```
%md

USA Flights 2008

http://stat-computing.org/dataexpo/2009/2008.csv.bz2
```

FINISHED

```
%md

First Example (for the impacients)
---

**Step by Step**: we will start just loading a csv file.
```

FINISHED

```
val localPath="/Users/juantomas/proyectos/cursos/curso-spark/datasets/"
val airports = sqlContext.read.format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(localPath+"airports.csv")
```

FINISHED

```
localPath: String = /Users/juantomas/proyectos/cursos/curso-spark/datasets/
airports: org.apache.spark.sql.DataFrame = [iata: string, airport: string ... 5 more field
s]
```

Took 3 sec. Last updated by anonymous at February 28 2017, 10:24:22 PM.

FINISHED

We just create a DataFrame reading the file

The schema of the Dataframe is:

Took 0 sec. Last updated by anonymous at February 28 2017, 10:28:51 PM.

```
%spark
airports.printSchema
```

FINISHED

```
root
 |-- iata: string (nullable = true)
 |-- airport: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- country: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
```

Took 1 sec. Last updated by anonymous at February 28 2017, 10:26:29 PM.

The contents of the dataframe airports is:

FINISHED

Took 0 sec. Last updated by anonymous at February 28 2017, 10:29:48 PM.

```
%spark

airports.show(10)
println("Total Airports: " + airports.count.toString)
```

FINISHED

```
+----+--------------------+---------------+-----+-------+-----------+------------+
|iata|             airport|           city|state|country|        lat|        long|
+----+--------------------+---------------+-----+-------+-----------+------------+
| 00M|            Thigpen |     Bay Springs|   MS|    USA|31.95376472|-89.23450472|
| 00R|Livingston Municipal|      Livingston|   TX|    USA|30.68586111|-95.01792778|
| 00V|         Meadow Lake|Colorado Springs|   CO|    USA|38.94574889|-104.5698933|
| 01G|        Perry-Warsaw|           Perry|   NY|    USA|42.74134667|-78.05208056|
| 01J|    Hilliard Airpark|        Hilliard|   FL|    USA| 30.6880125|-81.90594389|
| 01M|   Tishomingo County|         Belmont|   MS|    USA|34.49166667|-88.20111111|
| 02A|         Gragg-Wade |         Clanton|   AL|    USA|32.85048667|-86.61145333|
| 02C|             Capitol|      Brookfield|   WI|    USA|   43.08751|-88.17786917|
| 02G|   Columbiana County|  East Liverpool|   OH|    USA|40.67331278|-80.64140639|
| 03D|    Memphis Memorial|         Memphis|   MO|    USA|40.44725889|-92.22696056|
+----+--------------------+---------------+-----+-------+-----------+------------+
only showing top 10 rows
Total Airports: 3376
```

Took 3 sec. Last updated by anonymous at February 28 2017, 10:38:55 PM.

```
%spark
val flights = sqlContext.read.format("com.databricks.spark.csv")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(localPath+"2008.csv")
```

FINISHED

```
flights: org.apache.spark.sql.DataFrame = [Year: int, Month: int ... 27 more fields]
```

Took 31 sec. Last updated by anonymous at February 28 2017, 10:41:24 PM.

```
%spark
flights.printSchema
flights.show(10)
println("Toatal: " + flights.count)
```

FINISHED

```
root
 |-- Year: integer (nullable = true)
 |-- Month: integer (nullable = true)
 |-- DayofMonth: integer (nullable = true)
 |-- DayOfWeek: integer (nullable = true)
 |-- DepTime: string (nullable = true)
 |-- CRSDepTime: integer (nullable = true)
 |-- ArrTime: string (nullable = true)
 |-- CRSArrTime: integer (nullable = true)
 |-- UniqueCarrier: string (nullable = true)
 |-- FlightNum: integer (nullable = true)
 |-- TailNum: string (nullable = true)
 |-- ActualElapsedTime: string (nullable = true)
 |-- CRSElapsedTime: string (nullable = true)
 |-- AirTime: string (nullable = true)
 |-- ArrDelay: string (nullable = true)
 |-- DepDelay: string (nullable = true)
 |-- Origin: string (nullable = true)
```

Took 22 sec. Last updated by anonymous at February 28 2017, 10:42:58 PM. (outdated)

So what is Spark SQL?

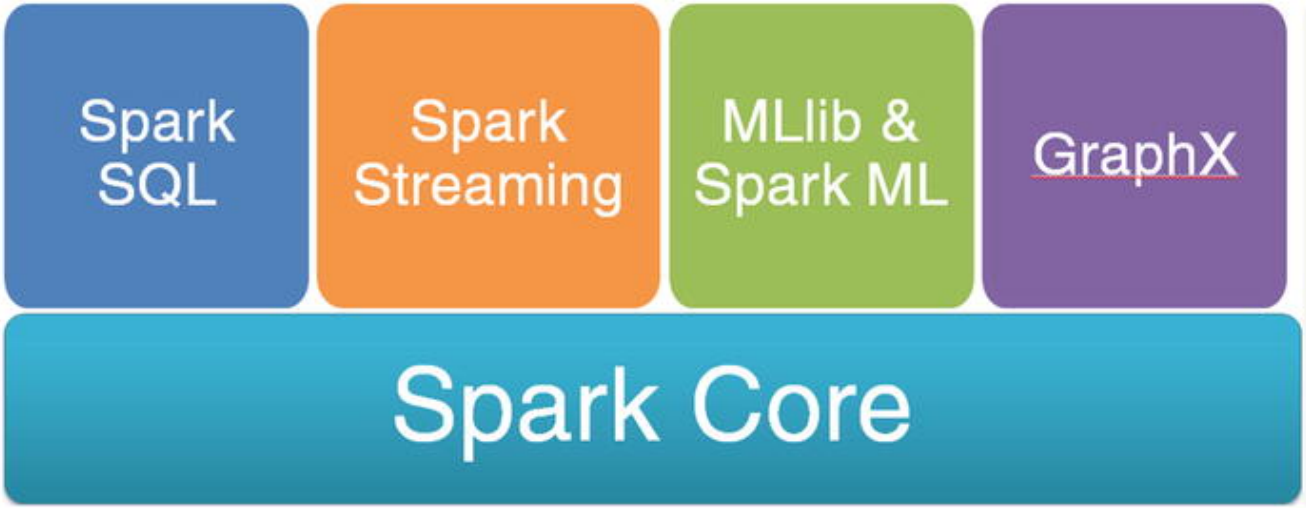
FINISHED

- At the very start was a derivate of HIVEQL
- Spark SQL is a Spark library that runs on top of Spark
- It provides a higher-level abstraction than the Spark core API for processing structured data
- Structured data includes data stored in a database, NoSQL data store, Parquet, ORC, Avro, JSON, CSV, or any other structured format
- The NoSQL data stores that can be used with Spark SQL include HBase, Cassandra, Elasticsearch, Druid, and other NoSQL data stores
- Spark SQL is more than just about providing SQL interface to Spark
- Spark SQL can be used as a library for developing data processing applications in Scala, Java, Python, or R
- internally uses the Spark core API to execute queries on a Spark cluster
- Spark SQL seamlessly integrates with other Spark libraries such as Spark Streaming, Spark ML, and GraphX

Took 0 sec. Last updated by anonymous at February 28 2017, 11:37:54 PM.

That’s the global view of spark & spark SQL

FINISHED



Took 0 sec. Last updated by anonymous at February 28 2017, 11:55:40 PM.

Keypoints of Spark SQL

FINISHED

Performance

TL;DR: Spark SQL makes data processing applications run faster using a combination of techniques, including reduced disk I/O, in-memory columnar caching, query optimization, and code generation.

- Reduced Disk I/O
  - Disk I/O is slow
  - can skip non-required partitions, rows, or columns while reading data.
- Partitioning
  - a lot of I/O can be avoided by partitioning a dataset
- Columnar Storage
  - Spark SQL supports columnar storage formats such as Parquet, which allow reading of only the columns that are used in a query
- In-Memory Columnar Caching
  - Spark SQL allows an application to cache data in an in-memory columnar format from any data source
  - Spark SQL, caches only the required columns
  - Spark SQL compresses the cached columns to minimize memory usage and JVM garbage collection pressure
  - apply efficient compression techniques such as run length encoding, delta encoding, and dictionary encoding
- Skip Rows
  - If a data source maintains statistical information about a dataset, Spark SQL takes advantage of it

    - Serialization formats such as Parquet and ORC store min and max values for each column in a row group or chunk of rows

- Predicate Pushdown
    - Instead of reading an entire table and then executing a filtering operation, Spark SQL will ask the database to natively execute the filtering operation
    - Since databases generally index data, native filtering is much faster than filtering at the application layer
- Query Optimization
    - analysis
    - logical optimization
    - physical planning
    - code generation
- Applications
    - Spark SQL can be used as a ETL
    - Very fast conversions between formats.
    - Few lines of code.
    - You can even perform join operations across different data sources.
- Distributed JDBC/ODBC SQL Query Engine
    - can be used as a library. In this mode, data processing tasks can be expressed as SQL, HiveQL or language integrated queries within a Scala, Java, Python, or R application
    - It comes prepackaged with a Thrift/JDBC/ODBC server
- Data Warehousing
    - Spark SQL store and analyze large amounts of data like a warehouse
    - Spark SQL-based data warehousing solution is more scalable, economical, and flexible than the proprietary data warehousing solutions
    - is flexible since it supports both schema-on-read and schema-on-write

Took 0 sec. Last updated by anonymous at March 01 2017, 12:18:31 AM.

Application Programming Interface (API)

FINISHED

The Spark SQL API consists of three key abstractions:

- SQLContext
- HiveContext
- DataFrame.

A Spark SQL application processes data using these abstractions.

Took 1 sec. Last updated by anonymous at March 01 2017, 12:20:33 AM.

SQLContext

FINISHED

SQLContext is the main entry point into the Spark SQL library.

It is a class defined in the Spark SQL library.

A Spark SQL application must create an instance of the SQLContext or HiveContext class.

How create it

```
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)

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

Took 0 sec. Last updated by anonymous at March 01 2017, 12:30:26 AM. (outdated)

FINISHED

```
%spark

sc
sqlContext
```

```
res11: org.apache.spark.SparkContext = org.apache.spark.SparkContext@3a63e394
res12: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@472983b3
```

Took 6 sec. Last updated by anonymous at March 01 2017, 12:25:09 AM.

DataFrames

FINISHED

- DataFrame is Spark SQL's primary data abstraction
- a distributed collection of rows organized into named columns.
- inspired by DataFrames in R and Python
- similar to a table in a relational database

Took 0 sec. Last updated by anonymous at March 01 2017, 12:33:13 AM.

FINISHED

```
%md

Diffs between RDDs & Dataframes
---

* DataFrame is schema aware
* RDD is a partitioned collection of opaque elements
* DataFrame knows the names and types of the columns in a dataset
* DataFrame class is able to provide a rich domain-specific-language (DSL) for data proces
```

```
* DataFrame API is easier to understand and use than the RDD API
* DataFrame can also be operated on as an RDD
* RDD can be easily created from a DataFrame
* DataFrame can be registered as a temporary table
```

Diffs between RDDs & Dataframes

- DataFrame is schema aware
- RDD is a partitioned collection of opaque elements
- DataFrame knows the names and types of the columns in a dataset
- DataFrame class is able to provide a rich domain-specific-language (DSL) for data processing
- DataFrame API is easier to understand and use than the RDD API
- DataFrame can also be operated on as an RDD
- RDD can be easily created from a DataFrame
- DataFrame can be registered as a temporary table

Took 0 sec. Last updated by anonymous at March 01 2017, 12:37:23 AM.

FINISHED

```
%md

Row
---

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

it is equivalent to a relational tuple or row in a table

You can create a Row this way:

~~~
import org.apache.spark.sql._

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

and you can use this rows at low level:

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

Row

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

it is equivalent to a relational tuple or row in a table

You can create a Row this way:

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

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

and you can use this rows at low level:

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

Took 0 sec. Last updated by anonymous at March 01 2017, 12:40:07 AM.

How to create a Dataframe:

FINISHED

a) from an 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()
```

b) 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)
```

c) Creating a DataFrame from a Data Source

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

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

the same API can be used to create a DataFrame from a Parquet, JSON, ORC or CSV file on local file system, HDFS or S3

Spark SQL has built-in support for some of the commonly used data sources

Includes Parquet, JSON, Hive, and JDBC-compliant databases

External packages are available for other 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")

// 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=<PAS
                  "dbtable" -> "schema-name.table-name"))
               .load()

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

Took 0 sec. Last updated by anonymous at March 01 2017, 12:50:52 AM.

```
%md
```

FINISHED

```
Some examples reading from different sources
---

**JSON**

~~~
val jsonDF = sqlContext.read.json("path/to/JSON-file-or-directory")
val jsonHdfsDF = sqlContext.read.json("hdfs://NAME_NODE/path/to/data.json")
val jsonS3DF = sqlContext.read.json("s3a://BUCKET_NAME/FOLDER_NAME/data.json")
~~~
Passing the schema to dataframe reader:
~~~
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")
~~~
```

Some examples reading from different sources

JSON

```
val jsonDF = sqlContext.read.json("path/to/JSON-file-or-directory")
val jsonHdfsDF = sqlContext.read.json("hdfs://NAME_NODE/path/to/data.json")
val jsonS3DF = sqlContext.read.json("s3a://BUCKET_NAME/FOLDER_NAME/data.json")
```

Passing the schema to dataframe reader:

```
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")
```

Took 0 sec. Last updated by anonymous at March 01 2017, 12:54:43 AM.

Parquet FINISHED

```
val parquetDF = sqlContext.read.parquet("path/to/parquet-file-or-directory")
```

ORC

```
val orcDF = hiveContext.read.orc("path/to/orc-file-or-directory")
```

Hive

```
val hiveDF = hiveContext.read.table("hive-table-name")
val hiveDF = hiveContext.sql("SELECT col_a, col_b, col_c from hive-table")
```

JDBC

```
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)
```

Took 0 sec. Last updated by anonymous at March 01 2017, 12:59:41 AM. (outdated)

Basic Operations

FINISHED

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")
```

columns

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)
```

dtypes

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,
```

explain

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
```

A variant of the explain method takes a Boolean argument and prints both logical and physical plans if the argument is true.

persist

The persist method caches the source DataFrame in memory.

```
customerDF.persist
```

Similar to the persist method in the RDD class, a variant of the persist method in the DataFrame class allows you to specify the storage level for a persisted DataFrame.

printSchema

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

The registerTempTable method creates a temporary table in Hive metastore. It takes a table name as an argument.

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]
```

Note that the sql method returns a DataFrame. Let's discuss how to view or retrieve the contents of a DataFrame in a moment.

toDF

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

Took 0 sec. Last updated by anonymous at March 01 2017, 1:25:11 AM.

```
%md                                                                    FINISHED
Language-Integrated Query Methods
---

**agg**

The agg method performs specified aggregations on one or more columns in the source DataFr
~~~
val aggregates = productDF.agg(max("price"), min("price"), count("name"))

aggregates: org.apache.spark.sql.DataFrame = [max(price): double, min(price): double, coun

aggregates.show
+----------+----------+-----------+
|max(price)|min(price)|count(name)|
+----------+----------+-----------+
|    1200.0|     200.0|          6|
+----------+----------+-----------+
~~~
Let's discuss the show method in a little bit. Essentially, it displays the content of a D

**apply**

The apply method takes the name of a column as an argument and returns the specified colum
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 provides syntactic sugar that allows you to use productDF("price") instead of produc
productDF("price") to productDF.apply("price"). So the preceding code can be rewritten, as

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

An instance of the Column class is generally used as an input to some of the DataFrame met
revisit one of the examples discussed earlier.
~~~
val aggregates = productDF.agg(max("price"), min("price"), count("name"))
~~~
It is a concise version of the statement shown next.
~~~
val aggregates = productDF.agg(max(productDF("price")), min(productDF("price")),
                               count(productDF("name")))
~~~
The expression productDF("price") can also be written as $"price" for convenience. Thus, t
~~~
val aggregates = productDF.agg(max($"price"), min($"price"), count($"name"))
val aggregates = productDF.agg(max(productDF("price")), min(productDF("price")),
                               count(productDF("name")))
~~~
If a method or function expects an instance of the Column class as an argument, you can us

In summary, 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"))
~~~
cube

The cube method takes the names of one or more columns as arguments and returns a cube for
cross-tabular reports.

Assume you have a dataset that tracks sales along three dimensions: time, product and coun
all the possible combinations of the dimensions that you are interested in.
~~~
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: str

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|
|01/01/2015| iPhone|    USA| 40000.0|
|01/01/2015|   null|  China| 19000.0|
|01/02/2015|   null|    USA| 40000.0|
|      null| iPhone|  China| 15000.0|
|01/02/2015|     S6|  China|  6000.0|
|01/01/2015| iPhone|  China| 10000.0|
|01/02/2015|   null|  China| 11000.0|
|      null| iPhone|   null| 85000.0|
|      null| iPhone|    USA| 70000.0|
|      null|     S6|   null| 45000.0|
|      null|     S6|    USA| 30000.0|
|01/01/2015|     S6|   null| 29000.0|
|      null|   null|   null|130000.0|
|01/02/2015| iPhone|   null| 35000.0|
|01/01/2015|     S6|    USA| 20000.0|
|      null|   null|    USA|100000.0|
|01/01/2015|   null|   null| 79000.0|
|      null|     S6|  China| 15000.0|
|01/02/2015| iPhone|    USA| 30000.0|
+----------+-------+-------+--------+
```

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

```
~~~
salesCubeDF.filter("product IS null AND date IS null AND country='USA'").show
~~~
+----+-------+-------+------------+
|date|product|country|sum(revenue)|
+----+-------+-------+------------+
|null|   null|    USA|    100000.0|
+----+-------+-------+------------+
```

If you wanted to know the subtotal of sales by product in the USA, you can use the followi

```
~~~
salesCubeDF.filter("date IS null AND product IS NOT null AND country='USA'").show
~~~
+----+-------+-------+------------+
|date|product|country|sum(revenue)|
+----+-------+-------+------------+
|null| iPhone|    USA|     70000.0|
|null|     S6|    USA|     30000.0|
+----+-------+-------+------------+
```

distinct

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

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

explode

The explode method generates zero or more rows from a column using a user-provided functio
input column, the second argument is the output column and the third argument is a user pr
output column for each value in the input column.

For example, consider a dataset that has a text column containing contents of an email. Le
individual words and you want 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
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:

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
~~~
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 i
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): doubl

revenueByProductDF.show

+-------+------------+
|product|sum(revenue)|
+-------+------------+
| iPhone|     85000.0|
|     S6|     45000.0|
+-------+------------+
~~~
**intersect**

The intersect method takes a DataFrame as an argument and returns a new DataFrame containi
~~~
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 tak
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 outerDF = transactionDF.join(customerDF, $"custId" === $"cId", "outer")
outerDF.show

+----+------+------+----------+-------------+----+--------+----+------+
| tId|custId|prodId|      date|         city| cId|    name| age|gender|
+----+------+------+----------+-------------+----+--------+----+------+
|   6|   100|   100|01/02/2015|Mountain View|null|    null|null|  null|
|   4|   200|   400|01/02/2015|    Palo Alto|null|    null|null|  null|
|   3|     1|     6|01/01/2015|       Boston|   1|   James|  21|     M|
|null|  null|  null|      null|         null|   2|     Liz|  25|     F|
|null|  null|  null|      null|         null|   3|    John|  31|     M|
|null|  null|  null|      null|         null|   4|Jennifer|  45|     F|
|   1|     5|     3|01/01/2015|San Francisco|   5|  Robert|  41|     M|
|   2|     6|     1|01/02/2015|     San Jose|   6|  Sandra|  45|     F|
+----+------+------+----------+-------------+----+--------+----+------+

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|
+---+------+------+----------+-------------+----+------+----+------+

val rightOuterDF = transactionDF.join(customerDF, $"custId" === $"cId", "right_outer")
rightOuterDF.show

+----+------+------+----------+-------------+---+--------+---+------+
| tId|custId|prodId|      date|         city|cId|    name|age|gender|
+----+------+------+----------+-------------+---+--------+---+------+
|   3|     1|     6|01/01/2015|       Boston|  1|   James| 21|     M|
|null|  null|  null|      null|         null|  2|     Liz| 25|     F|
|null|  null|  null|      null|         null|  3|    John| 31|     M|
|null|  null|  null|      null|         null|  4|Jennifer| 45|     F|
|   1|     5|     3|01/01/2015|San Francisco|  5|  Robert| 41|     M|
|   2|     6|     1|01/02/2015|     San Jose|  6|  Sandra| 45|     F|
+----+------+------+----------+-------------+---+--------+---+------+
~~~
**limit**

The limit method returns a DataFrame containing the specified number of rows from the sour
~~~
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|
+---+--------+---+------+
~~~
**orderBy**
```

The orderBy method returns a DataFrame sorted by the given columns. It takes the names of
~~~
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|
+---+--------+---+------+
~~~
By default, the orderBy method sorts in ascending order. You can explicitly specify the so
~~~
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

The randomSplit method splits the source DataFrame into multiple DataFrames. It takes an a
DataFrames. It is a useful method for machine learning, where you want to split the raw da
~~~
val dfArray = homeDF.randomSplit(Array(0.6, 0.2, 0.2))
dfArray(0).count
dfArray(1).count
dfArray(2).count
~~~
rollup

The rollup method takes the names of one or more columns as arguments and returns a multi-
hierarchical dimension such as geography or time.

Assume you have a dataset that tracks annual sales by city, state and country. The rollup
subtotals by city, state, and country.
~~~
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|
+-------+-----+---------+------------+
~~~
**sample**

The sample method returns a DataFrame containing the specified fraction of the rows in the
argument is a Boolean value indicating whether sampling should be done with replacement. T
should be returned.
~~~
val sampleDF = homeDF.sample(true, 0.10)
~~~
**select**

The select method returns a DataFrame containing only the specified columns from the sourc
~~~
val namesAgeDF = customerDF.select("name", "age")
namesAgeDF.show

+--------+---+
|    name|age|
+--------+---+
|   James| 21|
|     Liz| 25|
|    John| 31|
|Jennifer| 45|
|  Robert| 41|
|  Sandra| 45|
+--------+---+
~~~
A variant of the select method allows one or more Column expressions as arguments.
~~~
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 and returns a DataF
~~~
```

```
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 Da
arguments. The first argument is the name of the new column and the second argument is an
~~~
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|
+---+-------+------+-----+------+
~~~
```

Language-Integrated Query Methods

agg

The agg 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, cour

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

Let’s discuss the show method in a little bit. Essentially, it displays the content of a DataFrame on the console.

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 provides syntactic sugar that allows you to use productDF("price") instead of produc
```

val priceColumn = productDF(“price”)
val discountedPriceColumn = priceColumn * 0.5
~~~~

An instance of the Column class is generally used as an input to some of the DataFrame methods or functions defined in the Spark SQL library. Let’s revisit one of the examples discussed earlier.

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

It is a concise version of the statement shown next.

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

The expression productDF(“price”) can also be written as $“price” for convenience. Thus, the following two expressions are equivalent.

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

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.

In summary, 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"))
```
~~~~

cube

The cube method takes the names of one or more columns as arguments and 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 allows you to generate aggregates for all the possible combinations of the dimensions that you are interested in.

```
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: str

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|
|01/01/2015| iPhone| USA| 40000.0|
|01/01/2015| null| China| 19000.0|
|01/02/2015| null| USA| 40000.0|
| null| iPhone| China| 15000.0|
|01/02/2015| S6| China| 6000.0|
|01/01/2015| iPhone| China| 10000.0|
|01/02/2015| null| China| 11000.0|
| null| iPhone| null| 85000.0|
| null| iPhone| USA| 70000.0|
| null| S6| null| 45000.0|
| null| S6| USA| 30000.0|
|01/01/2015| S6| null| 29000.0|
| null| null| null|130000.0|
|01/02/2015| iPhone| null| 35000.0|
```

```
|01/01/2015| S6| USA| 20000.0|
| null| null| USA|100000.0|
|01/01/2015| null| null| 79000.0|
| null| S6| China| 15000.0|
|01/02/2015| iPhone| USA| 30000.0|
+————-+———-+———-+————+
```

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|
+—-+———-+———-+————————+
```

If you wanted to know the subtotal of sales by product in the USA, you can use the following expression.

```
salesCubeDF.filter("date IS null AND product IS NOT null AND country='USA'").show
```

```
+—-+———-+———-+————————+
|date|product|country|sum(revenue)|
+—-+———-+———-+————————+
|null| iPhone| USA| 70000.0|
|null| S6| USA| 30000.0|
+—-+———-+———-+————————+
```

distinct

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

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

explode

The explode method generates zero or more rows from a column using a user-provided function. It takes three arguments. The first argument is the input column, the second argument is the output column and the third argument is a user provided function that generates one or more values for the output column for each value in the input column.

For example, consider a dataset that has a text column containing contents of an email. Let's assume that you want to split the email content into individual words and you want 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:

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): doubl

revenueByProductDF.show

+-------+------------+
|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 outerDF = transactionDF.join(customerDF, $"custId" === $"cId", "outer")
outerDF.show

+----+------+------+----------+-------------+----+--------+----+------+
| tId|custId|prodId|      date|         city| cId|    name| age|gender|
+----+------+------+----------+-------------+----+--------+----+------+
|   6|   100|   100|01/02/2015|Mountain View|null|    null|null|  null|
|   4|   200|   400|01/02/2015|    Palo Alto|null|    null|null|  null|
|   3|     1|     6|01/01/2015|       Boston|   1|   James|  21|     M|
|null|  null|  null|      null|         null|   2|     Liz|  25|     F|
|null|  null|  null|      null|         null|   3|    John|  31|     M|
|null|  null|  null|      null|         null|   4|Jennifer|  45|     F|
|   1|     5|     3|01/01/2015|San Francisco|   5|  Robert|  41|     M|
|   2|     6|     1|01/02/2015|     San Jose|   6|  Sandra|  45|     F|
+----+------+------+----------+-------------+----+--------+----+------+

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|
+---+------+------+----------+-------------+----+------+----+------+

val rightOuterDF = transactionDF.join(customerDF, $"custId" === $"cId", "right_outer")
rightOuterDF.show

+----+------+------+----------+-------------+---+--------+---+------+
| tId|custId|prodId|      date|         city|cId|    name|age|gender|
+----+------+------+----------+-------------+---+--------+---+------+
|   3|     1|     6|01/01/2015|       Boston|  1|   James| 21|     M|
|null|  null|  null|      null|         null|  2|     Liz| 25|     F|
|null|  null|  null|      null|         null|  3|    John| 31|     M|
|null|  null|  null|      null|         null|  4|Jennifer| 45|     F|
|   1|     5|     3|01/01/2015|San Francisco|  5|  Robert| 41|     M|
```

```
|   2|     6|     1|01/02/2015|     San Jose|  6|  Sandra| 45|     F|
+----+------+------+----------+-------------+---+--------+---+------+
```

limit

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

orderBy

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

By default, the orderBy method sorts in ascending order. You can explicitly specify the sorting order using a Column expression, as shown next.

```
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

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
```

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.

Assume you have a dataset that tracks annual sales by city, state and country. The rollup method can be used to calculate both grand total and subtotals by city, state, and country.

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

sample

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)
```

select

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

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

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

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

```
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 and 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 first argument is the name of the new column and the second argument is 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|
+---+-------+------+-----+------+
```

Took 0 sec. Last updated by anonymous at March 01 2017, 1:44:21 AM.

RDD Operations

FINISHED

The DataFrame class supports commonly used RDD operations such as map, flatMap, foreach, foreachPartition, mapPartition, coalesce, and repartition. These methods work similar to their namesake operations in the RDD class.

In addition, if you need access to other RDD methods that are not present in the DataFrame class, you can get an RDD from a DataFrame. This section discusses the commonly used techniques for generating an RDD from a DataFrame.

rdd

rdd is defined as a lazy val in the DataFrame class. It represents the source DataFrame as an RDD of Row instances.

As discussed earlier, a Row represents a relational tuple in the source DataFrame. It allows both generic access and native primitive access of fields by their ordinal.

An example is shown next.

```
val rdd = customerDF.rdd

rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[405] at rdd at

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
```

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

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

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

nameAndAge.collect

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

toJSON

The toJSON method generates an RDD of JSON strings from the source DataFrame. Each element in the returned RDD is a JSON object.

```
val jsonRDD = customerDF.toJSON

jsonRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[408] at toJSON at <console>:2

jsonRDD.collect

res80: Array[String] = Array({"cId":1,"name":"James","age":21,"gender":"M"}, {"cId":2,"nam
```

Actions

Similar to the RDD actions, the action methods in the DataFrame class return results to the Driver program. This section covers the commonly used action methods in the DataFrame class.

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
```

count

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

```
val count = customerDF.count

count: Long = 6
```

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. It takes the names of one or more columns as arguments.

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

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

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

first

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]
```

show

The show method displays the rows in the source DataFrame on the driver console in a tabular format. It optionally takes an integer N as an argument and displays the top N rows. If no argument is provided, it shows the top 20 rows.

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

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

take

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])
```

Took 0 sec. Last updated by anonymous at March 01 2017, 2:05:44 AM. (outdated)

```
%md
Output Operations
---
An output operation saves a DataFrame to a storage system. Prior to version 1.4, DataFrame
DataFrame to a variety of storage systems. Starting with version 1.4, those methods were r

**write**

The write method returns an instance of the DataFrameWriter class, which provides methods
The next section covers the DataFrameWriter class.

**Saving a DataFrame**

Spark SQL provides a unified interface for saving a DataFrame to a variety of data sources
relational databases, NoSQL data stores and a variety of file formats.

The DataFrameWriter class defines the interface for writing data to a data source. Through
options for saving data. For example, you can specify format, partitioning, and handling o

The following examples show how to save a DataFrame to different storage systems.
~~~
// 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")
```

FINISHED

```
// 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")
~~~
You can save a DataFrame in Parquet, JSON, ORC, or CSV format to any Hadoop-supported stor

If a data source supports partitioned layout, the DataFrameWriter class supports it throug
specified column and create a separate subdirectory for each unique value in the specified
during query. Future queries through Spark SQL will be able to skip large amounts of disk

Consider the following example.
~~~
homeDF.write
  .format("parquet")
  .partitionBy("city")
  .save("homes")
~~~
The preceding code partitions rows by the city column. A subdirectory will be created for
named city=Berkeley, city=Fremont, city=Oakland, and so on, will be created under the home

The following code will only read the subdirectory named city=Berkeley and skip all other
dataset that includes data for hundreds of cities, this could speed up application perform
~~~
val newHomesDF = sqlContext.read.format("parquet").load("homes")
newHomesDF.registerTempTable("homes")
val homesInBerkeley = sqlContext.sql("SELECT * FROM homes WHERE city = 'Berkeley'")
~~~
While saving a DataFrame, if the destination path or table already exists, Spark SQL will
by calling the mode method in the DataFrameWriter class. It takes an argument, which speci
exists. The mode method supports the following options:
~~~
    error (default) - throw an exception if destination path/table already exists.
    append - append to existing data if destination path/table already exists.
    overwrite - overwrite existing data if destination path/table exists.
    ignore - ignore the operation if destination path/table exists.
~~~
A few examples are shown next.
~~~
customerDF.write
  .format("parquet")
  .mode("overwrite")
  .save("path/to/output-directory")

customerDF.write
  .mode("append")
  .saveAsTable("hive-table-name")
~~~
In addition to the methods shown here for writing data to a data source, the DataFrameWrit
data sources for which it has built-in support for. These data sources include Parquet, OR

**JSON**

The json method saves the contents of a DataFrame in JSON format. It takes as argument a p
~~~
customerDF.write.json("path/to/directory")
~~~
**Parquet**
```

```
The parquet method saves the contents of a DataFrame in Parquet format. It takes a path as
~~~
customerDF.write.parquet("path/to/directory")
~~~
**ORC**

The orc method saves a DataFrame in ORC file format. Similar to the JSON and Parquet metho
~~~
customerDF.write.orc("path/to/directory")
~~~
**Hive**

The saveAsTable method saves the content of a DataFrame as a Hive table. It saves a DataFr
metastore.
~~~
customerDF.write.saveAsTable("hive-table-name")
~~~
**JDBC-Compliant Database**

The jdbc method saves a DataFrame to a database using the JDBC interface. It takes three a
connection properties. The connection properties specify connection arguments such as user
~~~
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")

customerDF.write.jdbc(jdbcUrl, tableName, connectionProperties)

~~~
```

Output Operations

An output operation saves a DataFrame to a storage system. Prior to version 1.4, DataFrame included a number of different methods for saving a DataFrame to a variety of storage systems. Starting with version 1.4, those methods were replaced by the write method.

write

The write method returns an instance of the DataFrameWriter class, which provides methods for saving the contents of a DataFrame to a data source. The next section covers the DataFrameWriter class.

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. Through its builder methods, it allows you to specify different options for saving data. For example, you can specify format, partitioning, and handling of existing data.

The following examples show how to save a DataFrame to different storage systems.

```
// 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")
```

You can save a DataFrame in Parquet, JSON, ORC, or CSV format to any Hadoop-supported storage system, including local file system, HDFS or Amazon S3.

If a data source supports partitioned layout, the DataFrameWriter class supports it through the partitionBy method. It will partition the rows by the specified column and create a separate subdirectory for each unique value in the specified column. Partitioned layout enables partition pruning during query. Future queries through Spark SQL will be able to skip large amounts of disk I/O when a partitioned column is referenced in a predicate.

Consider the following example.

```
homeDF.write
  .format("parquet")
  .partitionBy("city")
  .save("homes")
```

The preceding code partitions rows by the city column. A subdirectory will be created for each unique value of city. For example, subdirectories named city=Berkeley, city=Fremont, city=Oakland, and so on, will be created under the homes directory.

The following code will only read the subdirectory named city=Berkeley and skip all other subdirectories under the homes directory. For a large dataset that includes data for hundreds of cities, this could speed up application performance by orders of magnitude.

```
val newHomesDF = sqlContext.read.format("parquet").load("homes")
newHomesDF.registerTempTable("homes")
val homesInBerkeley = sqlContext.sql("SELECT * FROM homes WHERE city = 'Berkeley'")
```

While saving a DataFrame, if the destination path or table already exists, Spark SQL will throw an exception by default. You can change this behavior by calling the mode method in the DataFrameWriter class. It takes an argument, which specifies the behavior if destination path or table already exists. The mode method supports the following options:

```
    error (default) - throw an exception if destination path/table already exists.
    append - append to existing data if destination path/table already exists.
    overwrite - overwrite existing data if destination path/table exists.
    ignore - ignore the operation if destination path/table exists.
```

A few examples are shown next.

```
customerDF.write
  .format("parquet")
  .mode("overwrite")
  .save("path/to/output-directory")

customerDF.write
  .mode("append")
  .saveAsTable("hive-table-name")
```

In addition to the methods shown here for writing data to a data source, the DataFrameWriter class provides special methods for writing data to the data sources for which it has built-in support for. These data sources include Parquet, ORC, JSON, Hive, and JDBC-compliant databases.

JSON

The json method saves the contents of a DataFrame in JSON format. It takes as argument a path, which can be on a local file system, HDFS or S3.

```
customerDF.write.json("path/to/directory")
```

Parquet

The parquet method saves the contents of a DataFrame in Parquet format. It takes a path as argument and saves a DataFrame at the specified path.

```
customerDF.write.parquet("path/to/directory")
```

ORC

The orc method saves a DataFrame in ORC file format. Similar to the JSON and Parquet methods, it takes a path as argument.

```
customerDF.write.orc("path/to/directory")
```

Hive

The saveAsTable method saves the content of a DataFrame as a Hive table. It saves a DataFrame to a file and registers metadata as a table in Hive metastore.

```
customerDF.write.saveAsTable("hive-table-name")
```

JDBC-Compliant Database

The jdbc method saves a DataFrame to a database using the JDBC interface. It takes three arguments: JDBC URL of a database, table name, and connection properties. The connection properties specify connection arguments such as user name and password.

```
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")

customerDF.write.jdbc(jdbcUrl, tableName, connectionProperties)
```

Took 1 sec. Last updated by anonymous at March 01 2017, 2:11:24 AM.

Built-in Functions

FINISHED

Spark SQL comes with a comprehensive list of built-in functions, which are optimized for fast execution. It implements these functions with code generation techniques. 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.

The following example illustrates how you can use a built-in function.

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

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

Collection

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

Date/Time

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

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. Conversely, the from_unixtime function converts a Unix epoch value to a string representation. Spark SQL also provides functions for converting timestamps from one time zone to another.

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

Field Extraction

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.

Date Arithmetic

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

In addition to the functions mentioned earlier, Spark SQL provides a few other useful date- and time-related functions, such as current_date, current_timestamp, trunc, date_format.

Math

The math functions operate on columns containing numerical values. Spark SQL comes with a long list of built-in math functions. Example include abs, ceil, cos, exp, factorial, floor, hex, hypot, log, log10, pow, round, shiftLeft, sin, sqrt, tan, and other commonly used math functions.

String

Spark SQL provides a variety of built-in functions for processing columns that contain string values. For example, you can split, trim or change case of a string. 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.

Window

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.

Took 1 sec. Last updated by anonymous at March 01 2017, 2:15:06 AM.

UDFs and UDAFs

FINISHED

Spark SQL allows user-defined functions (UDFs) and user-defined aggregation functions (UDAFs). Both UDFs and UDAFs perform custom computations on a dataset. A UDF performs custom computation one row at a time and returns a value for each row. A UDAF applies custom aggregation over groups of rows.

UDFs and UDAFs can be used just like the built-in functions after they have been registered with Spark SQL.

Writing a UDF is as easy as:

```
def setupUDFs(sqlCtx: SQLContext) = {
  sqlCtx.udf.register("strLen", (s: String) => s.length())
}
```

An UDAF is a little tricky:

```
def setupUDAFs(sqlCtx: SQLContext) = {
  class Avg extends UserDefinedAggregateFunction {
    // Input type
    def inputSchema: org.apache.spark.sql.types.StructType =
      StructType(StructField("value", DoubleType) :: Nil)

    def bufferSchema: StructType = StructType(
      StructField("count", LongType) ::
      StructField("sum", DoubleType) :: Nil
    )

    // Return type
    def dataType: DataType = DoubleType

    def deterministic: Boolean = true

    def initialize(buffer: MutableAggregationBuffer): Unit = {
      buffer(0) = 0L
      buffer(1) = 0.0
    }

    def update(buffer: MutableAggregationBuffer,input: Row): Unit = {
      buffer(0) = buffer.getAs[Long](0) + 1
      buffer(1) = buffer.getAs[Double](1) + input.getAs[Double](0)
    }

    def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
      buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
      buffer1(1) = buffer1.getAs[Double](1) + buffer2.getAs[Double](1)
    }

    def evaluate(buffer: Row): Any = {
      buffer.getDouble(1) / buffer.getLong(0)
    }
  }
  // Optionally register
  val avg = new Avg
  sqlCtx.udf.register("ourAvg", avg)
}
```

Interactive Analysis Example

As you have seen in previous sections, Spark SQL can be used as an interactive data analysis tool. The previous sections used toy examples so that they were easy to understand. This section shows how Spark SQL can be used for interactive data analysis on a real-world dataset. You can use either language integrated queries or SQL/HiveQL for data analysis. This section demonstrates how they can be used interchangeably.

The dataset that will be used is the Yelp challenge dataset. You can download it from www.yelp.com/dataset_challenge. It includes a number of data files. The example uses the yelp_academic_dataset_business.json file, which contains information about businesses. It contains a JSON object per line. Each JSON object contains information about a business, including its name, city, state, reviews, average rating, category and other attributes. You can learn more details about the dataset on Yelp's website.

Let's launch the Spark shell from a terminal, if it is not already running.

```
path/to/spark/bin/spark-shell --master local[*]
```

As discussed previously, the –master argument specifies the Spark master to which the Spark shell should connect. For the examples in this section, you can use Spark in local mode. If you have a real Spark cluster, change the master URL argument to point to your Spark master. The rest of the code in this section does not require any change.

For readability, I have split some of the example code statements into multiple lines. However, the Spark shell executes a statement as soon as you press the ENTER key. For multiline code statements, you can use Spark shell's paste mode (:paste). Alternatively, enter a complete statement on a single line.

Consider the following example.

```
biz.filter("...")
   .select("...")
   .show()
```

In the Spark shell, type it without the line breaks, as shown next.

```
biz.filter("...").select("...").show()
```

Consider another example.

```
sqlContext.sql("SELECT x, count(y) as total FROM t
                GROUP BY x
                ORDER BY total")
           .show(50)
```

Type the preceding code in the Spark shell without line breaks, as shown next.

```
sqlContext.sql("SELECT x, count(y) as total FROM t GROUP BY x ORDER BY total ").show(50)
```

Let's get started now.

Since you will be using a few classes and functions from the Spark SQL library, you need the following import statement.

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

Let's create a DataFrame from the Yelp businesses dataset.

```
val biz = sqlContext.read.json("path/to/yelp_academic_dataset_business.json")
```

The preceding statement is equivalent to this:

```
val biz = sqlContext.read.format("json").load("path/to/yelp_academic_dataset_business.jsor
```

Make sure that you specify the correct file path. Replace “path/to” with the name of the directory where you unpacked the Yelp dataset. Spark SQL will throw an exception if it cannot find the file at the specified path.

Spark SQL reads the entire dataset once to infer the schema from a JSON file. Let’s check the schema inferred by Spark SQL.

```
biz.printSchema()

Partial output is shown next.

root
 |-- attributes: struct (nullable = true)
 |    |-- Accepts Credit Cards: string (nullable = true)
 |    |-- Ages Allowed: string (nullable = true)
 |    |-- Alcohol: string (nullable = true)
 ...
 ...
 ...
 |-- name: string (nullable = true)
 |-- open: boolean (nullable = true)
 |-- review_count: long (nullable = true)
 |-- stars: double (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- type: string (nullable = true)
```

To use the SQL/HiveQL interface, you need to register the biz DataFrame as a temporary table.

```
biz.registerTempTable("biz")
```

Now you can analyze the Yelp businesses dataset using either the DataFrame API or SQL/HiveQL. I will show the code for both, but show the output only for the SQL/HiveQL queries.

Let’s cache the data in memory since you will be querying it more than one once.

Language-Integrated Query

```
biz.cache()
```

SQL

```
sqlContext.cacheTable("biz")
```

You can also cache a table using the CACHE TABLE statement, as shown next.

```
sqlContext.sql("CACHE TABLE biz")
```

Note that unlike RDD caching, Spark SQL immediately caches a table when you use the CACHE TABLE statement.

Since the goal of this section is to just demonstrate how you can use the Spark SQL library for interactive data analysis, the steps from this point onward have no specific ordering. I chose a few seemingly random queries. You may analyze data in a different sequence. Before each example, I will state the goal and then show the Spark SQL code for achieving that goal.

Let's find the number of businesses in the dataset.

Language-Integrated Query

```
val count = biz.count()
```

SQL

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

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

Next, let's find the count of businesses by state.

Language-Integrated Query

```
val bizCountByState = biz.groupBy("state").count
bizCountByState.show(50)
```

SQL

```
sqlContext.sql("SELECT state, count(1) as businesses FROM biz GROUP BY state").show(50)

+-----+----------+
|state|businesses|
+-----+----------+
|  XGL|         1|
|   NC|      4963|
|   NV|     16485|
|   AZ|     25230|
...
...
|  MLN|       123|
|  NTH|         1|
|   MN|         1|
+-----+----------+
```

Next, let's find the count of businesses by state and sort the result by count in descending order.

Language-Integrated Query

```
val resultDF = biz.groupBy("state").count
resultDF.orderBy($"count".desc).show(5)
```

SQL

```
sqlContext.sql("SELECT state, count(1) as businesses FROM biz GROUP BY state ORDER BY busi

+-----+-------------+
|state|businesses   |
+-----+-------------+
|   AZ|        25230|
|   NV|        16485|
|   NC|         4963|
|   QC|         3921|
|   PA|         3041|
+-----+-------------+
```

only showing top 5 rows

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

```
val resultDF = biz.groupBy("state").count
resultDF.orderBy(resultDF("count").desc).show(5)
```

The difference between the first and second version is how you refer to the count column in the resultDF DataFrame. You can think of $"count" as a shortcut for resultDF("count"). If a method or a function expects an argument of type Column, you can use either syntax.

Next, let's find five businesses with a five-star rating.

Language-Integrated Query

```
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. The <=> operator does equality test that is safe for null values. After filtering the businesses, it selects a subset of the columns that are of interest. Finally, it displays five businesses on the console.

SQL

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

+--------------------+-----+------------+------------+-----+
|                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|
+--------------------+-----+------------+------------+-----+
only showing top 5 rows
```

For the next example, let’s find three businesses with five stars in Nevada.

Language Integrated Query

```
biz.filter($"stars" <=> 5.0 && $"state" <=> "NV")
   .select("name","stars", "review_count", "city",  "state")
   .show(3)
```

SQL

```
sqlContext.sql("SELECT name, stars, review_count, city, state FROM biz WHERE state = 'NV'

+--------------------+-----+------------+---------+-----+
|                name|stars|review_count|     city|state|
+--------------------+-----+------------+---------+-----+
|              Adiamo|  5.0|           4|Henderson|   NV|
|CD Young's Profes...|  5.0|           8|Henderson|   NV|
|Liaisons Salon & Spa|  5.0|           5|Henderson|   NV|
+--------------------+-----+------------+---------+-----+
only showing top 3 rows
```

Next, let’s find the total number of reviews in each state.

Language Integrated Query

```
biz.groupBy("state").sum("review_count").show()
```

SQL

```
sqlContext.sql("SELECT state, sum(review_count) as reviews FROM biz GROUP BY state").show(

+-----+-------+
|state|reviews|
+-----+-------+
|  XGL|      3|
|   NC| 102495|
|   NV| 752904|
|   AZ| 636779|
...
...
|   QC|  54569|
|  KHL|      8|
|   RP|     75|
+-----+-------+
only showing top 20 rows
```

Next, let's find count of businesses by stars.

Language Integrated Query

```
biz.groupBy("stars").count.show()
```

SQL

```
sqlContext.sql("SELECT stars, count(1) as businesses FROM biz GROUP BY stars").show()

+-----+----------+
|stars|businesses|
+-----+----------+
|  1.0|       637|
|  3.5|     13171|
|  4.5|      9542|
|  3.0|      8335|
|  1.5|      1095|
|  5.0|      7354|
|  2.5|      5211|
|  4.0|     13475|
|  2.0|      2364|
+-----+----------+
```

Next, let's find the average number of reviews for a business by state.

Language-Integrated Query

```
val avgReviewsByState = biz.groupBy("state").avg("review_count")
avgReviewsByState.show()
```

SQL

```
sqlContext.sql("SELECT state, AVG(review_count) as avg_reviews FROM biz GROUP BY state").s

+-----+------------------+
|state|       avg_reviews|
+-----+------------------+
|  XGL|               3.0|
|   NC|20.651823493854522|
|   NV| 45.67206551410373|
|   AZ|25.238961553705906|
...
...
|   QC|13.917112981382301|
|  KHL|               8.0|
|   RP| 5.769230769230769|
+-----+------------------+
only showing top 20 rows
```

Next, let's find the top five states by the average number of reviews for a business.

Language Integrated Query

```
biz.groupBy("state")
   .avg("review_count")
   .withColumnRenamed("AVG(review_count)", "rc")
   .orderBy($"rc".desc)
   .selectExpr("state", "ROUND(rc) as avg_reviews")
   .show(5)
```

SQL

```
sqlContext.sql("SELECT state, ROUND(AVG(review_count)) as avg_reviews FROM biz GROUP BY st

+-----+-----------+
|state|avg_reviews|
+-----+-----------+
|   NV|       46.0|
|   AZ|       25.0|
|   PA|       24.0|
|   NC|       21.0|
|   IL|       21.0|
+-----+-----------+
```

Next, let's find the top 5 businesses in Las Vegas by average stars and review counts.

Language Integrated Query

```
biz.filter($"city" === "Las Vegas")
   .sort($"stars".desc, $"review_count".desc)
   .select($"name", $"stars", $"review_count")
   .show(5)
```

SQL

```
sqlContext.sql("SELECT name, stars, review_count FROM biz WHERE city = 'Las Vegas' ORDER B
```

```
+--------------------+-----+------------+
|                name|stars|review_count|
+--------------------+-----+------------+
|      Art of Flavors|  5.0|         321|
|Free Vegas Club P...|  5.0|         285|
|Fabulous Eyebrow ...|  5.0|         244|
|Raiding The Rock ...|  5.0|         199|
|            Eco-Tint|  5.0|         193|
+--------------------+-----+------------+
```

Next, let's write the data in Parquet format.

```
biz.write.mode("overwrite").parquet("path/to/yelp_business.parquet")
```

You can read the Parquet files created in the previous step, as shown next.

```
val ybDF = sqlContext.read.parquet("path/to/yelp_business.parquet")
```

Took 1 sec. Last updated by anonymous at March 01 2017, 2:47:43 AM.

Interactive Analysis with Spark SQL JDBC Server

FINISHED

This section shows how you can explore the Yelp dataset using just SQL/HiveQL. Scala is not used at all. For this analysis, you will use the Spark SQL Thrift/JDBC/ODBC server and the Beeline client. Both come prepackaged with Spark.

The first step is to launch the Spark SQL Thrift/JDBC/ODBC server from a terminal. Spark's sbin directory contains a script for launching it.

```
path/to/spark/sbin/start-thriftserver.sh --master local[*]
```

The second step is to launch Beeline, which is a CLI (command line interface) client for Spark SQL Thrift/JDBC server. Conceptually, it is similar to the mysql client for MySQL or psql client for PostgreSQL. It allows you to type a HiveQL query, sends the typed query to a Spark SQL Thrift/JDBC server for execution, and displays the results on the console.

Spark's bin directory contains a script for launching Beeline. Let's open another terminal and launch Beeline.

```
path/to/spark/bin/beeline
```

You should be now inside the Beeline shell and see the Beeline prompt, as shown next.

```
Beeline version 1.5.2 by Apache Hive
beeline>
```

The third step is to connect to the Spark SQL Thrift/JDBC server from the Beeline shell.

```
beeline>!connect jdbc:hive2://localhost:10000
```

The connect command requires a JDBC URL. The default JDBC port for the Spark SQL Thrift/JDBC server is 10000.

Beeline will ask for a username and password. Enter the username that you use to login into your system and a blank password.

```
beeline> !connect jdbc:hive2://localhost:10000

scan complete in 26ms
Connecting to jdbc:hive2://localhost:10000
Enter username for jdbc:hive2://localhost:10000: your-user-name
Enter password for jdbc:hive2://localhost:10000:
Connected to: Spark SQL (version 1.5.2)
Driver: Spark Project Core (version 1.5.2)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000>
```

At this point, you have an active connection between the Beeline client and Spark SQL server.

However, the Spark SQL server does not yet know about the Yelp dataset. Let's create a temporary table that points to the Yelp dataset.

```
0: jdbc:hive2://localhost:10000> CREATE TEMPORARY TABLE biz USING org.apache.spark.sql.jso
```

Make sure to replace "path/to" with the path for the directory where you unpacked the Yelp dataset on your machine. Spark SQL will throw an exception if it cannot find the file at the specified path.

The CREATE TEMPORARY TABLE command creates an external temporary table in Hive metastore. A temporary table exists only while the Spark SQL JDBC server is running.

Now you can analyze the Yelp dataset using just plain SQL/HiveQL queries. A few examples are shown next.

```
0: jdbc:hive2://localhost:10000> SHOW TABLES;

+------------+-------------+--+
| tableName  | isTemporary  |
+------------+-------------+--+
| biz        | true         |
+------------+-------------+--+
```

```
0: jdbc:hive2://localhost:10000> SELECT count(1) from biz;

+--------+--+
|  _c0   |
+--------+--+
| 61184  |
+--------+--+
```

```
0: jdbc:hive2://localhost:10000> SELECT state, count(1) as cnt FROM biz GROUP BY state ORD

+--------+--------+--+
| state  |  cnt   |
+--------+--------+--+
| AZ     | 25230  |
| NV     | 16485  |
| NC     | 4963   |
| QC     | 3921   |
| PA     | 3041   |
+--------+--------+--+
5 rows selected (3.241 seconds)
```

```
0: jdbc:hive2://localhost:10000> SELECT state, count(1) as businesses, sum(review_count) a

+--------+-------------+----------+--+
| state  | businesses  | reviews  |
+--------+-------------+----------+--+
| AZ     | 25230       | 636779   |
| NV     | 16485       | 752904   |
| NC     | 4963        | 102495   |
| QC     | 3921        | 54569    |
| PA     | 3041        | 72409    |
+--------+-------------+----------+--+
5 rows selected (1.293 seconds)
```

```
0: jdbc:hive2://localhost:10000> SELECT name, review_count, stars, city, state from biz WH

+---------------------------+--------------+--------+-------------+--------+--+
|           name            | review_count  | stars  |    city     | state  |
+---------------------------+--------------+--------+-------------+--------+--+
| Art of Flavors            | 321           | 5.0    | Las Vegas   | NV     |
| PNC Park                  | 306           | 5.0    | Pittsburgh  | PA     |
| Gaucho Parrilla Argentina | 286           | 5.0    | Pittsburgh  | PA     |
| Free Vegas Club Passes    | 285           | 5.0    | Las Vegas   | NV     |
| Little Miss BBQ           | 267           | 5.0    | Phoenix     | AZ     |
+---------------------------+--------------+--------+-------------+--------+--+
5 rows selected (0.511 seconds)
```

Took 0 sec. Last updated by anonymous at March 01 2017, 2:55:15 AM. (outdated)

Here is the APIs comparison between RDD, Dataframe and Dataset.

FINISHED

RDD

The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

RDD Features:

- Distributed collection:
  RDD uses MapReduce operations which is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

- Immutable: RDDs composed of a collection of records which are partitioned. A partition is a basic unit of parallelism in an RDD, and each partition is one logical division of data which is immutable and created through some transformations on existing partitions.Immutability helps to achieve consistency in computations.

- Fault tolerant: In a case of we lose some partition of RDD , we can replay the transformation on that partition in lineage to achieve the same computation, rather than doing data replication across multiple nodes.This characteristic is the biggest benefit of RDD because it saves a lot of efforts in data management and replication and thus achieves faster computations.

- Lazy evaluations: All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset . The transformations are only computed when an action requires a result to be returned to the driver program.

- Functional transformations: RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.

- Data processing formats:
  It can easily and efficiently process data which is structured as well as unstructured data.

*Programming Languages supported:
RDD API is available in Java, Scala, Python and R.

RDD Limitations:

- No inbuilt optimization engine: When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including catalyst optimizer and Tungsten execution engine. Developers need to optimize each RDD based on its attributes.

- Handling structured data: Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

Dataframes

Spark introduced Dataframes in Spark 1.3 release. Dataframe overcomes the key challenges that RDDs had.

A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a R/Python Dataframe. Along with Dataframe, Spark also introduced catalyst optimizer, which leverages advanced programming features to build an extensible query optimizer.

** Dataframe Features:**

- Distributed collection of Row Object: A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database, but with richer optimizations under the hood.

- Data Processing: Processing structured and unstructured data formats (Avro, CSV, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, MySQL, etc). It can read and write from all these various datasources.

- Optimization using catalyst optimizer: It powers both SQL queries and the DataFrame API. Dataframe use catalyst tree transformation framework in four phases,

  1.Analyzing a logical plan to resolve references
  2.Logical plan optimization
  3.Physical planning
  4.Code generation to compile parts of the query to Java bytecode.

- Hive Compatibility: Using Spark SQL, you can run unmodified Hive queries on your existing Hive warehouses. It reuses Hive frontend and MetaStore and gives you full compatibility with existing Hive data, queries, and UDFs.

- Tungsten: Tungsten provides a physical execution backend whichexplicitly manages memory and dynamically generates bytecode for expression evaluation.

- Programming Languages supported:
  Dataframe API is available in Java, Scala, Python, and R.

Dataframe Limitations:

Compile-time type safety: As discussed, Dataframe API does not support compile time safety which limits you from manipulating data when the structure is not know. The following example works during compile time. However, you will get a Runtime exception when executing this code.

Example:

```
case class Person(name : String , age : Int)
val dataframe = sqlContect.read.json("people.json")
dataframe.filter("salary > 10000").show
=> throws Exception : cannot resolve 'salary' given input age , name
```

This is challenging specially when you are working with several transformation and aggregation steps.

Cannot operate on domain Object (lost domain object): Once you have transformed a domain object into dataframe, you cannot regenerate it from it. In the following example, once we have create personDF from personRDD, we won't be recover the original RDD of Person class (RDD[Person]).

Example:

```
case class Person(name : String , age : Int)
val personRDD = sc.makeRDD(Seq(Person("A",10),Person("B",20)))
val personDF = sqlContect.createDataframe(personRDD)
personDF.rdd // returns RDD[Row] , does not returns RDD[Person]
```

Datasets API

Dataset API is an extension to DataFrames that provides a type-safe, object-oriented programming interface. It is a strongly-typed, immutable collection of objects that are mapped to a relational schema.

At the core of the Dataset, API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation. The tabular representation is stored using Spark internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization. Spark 1.6 comes with support for automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans.

Dataset Features:

- Provides best of both RDD and Dataframe: RDD(functional programming, type safe), DataFrame (relational model, Query optimazation , Tungsten execution, sorting and shuffling)
- Encoders: With the use of Encoders, it is easy to convert any JVM object into a Dataset, allowing users to work with both structured and unstructured data unlike Dataframe.

- Programming Languages supported: Datasets API is currently only available in Scala and Java. Python and R are currently not supported in version 1.6. Python support is slated for version 2.0.

- Type Safety: Datasets API provides compile time safety which was not available in Dataframes. In the example below, we can see how Dataset can operate on domain objects with compile lambda functions.

Example:

```
case class Person(name : String , age : Int)
val personRDD = sc.makeRDD(Seq(Person("A",10),Person("B",20)))
val personDF = sqlContect.createDataframe(personRDD)
val ds:Dataset[Person] = personDF.as[Person]
ds.filter(p => p.age > 25)
ds.filter(p => p.salary > 25)
 // error : value salary is not a member of person
ds.rdd // returns RDD[Person]
```

- Interoperable: Datasets allows you to easily convert your existing RDDs and Dataframes into datasets without boilerplate code.

Datasets API Limitation:

Requires type casting to String: Querying the data from datasets currently requires us to specify the fields in the class as a string. Once we have queried the data, we are forced to cast column to the required data type. On the other hand, if we use map operation on Datasets, it will not use Catalyst optimizer.

Example:

```
ds.select(col("name").as[String], $"age".as[Int]).collect()
```

No support for Python and R: As of release 1.6, Datasets only support Scala and Java. Python support will be introduced in Spark 2.0.

The Datasets API brings in several advantages over the existing RDD and Dataframe API with better type safety and functional programming.With the challenge of type casting requirements in the API, you would still not the required type safety and will make your code brittle.
shareimprove this answer

Took 0 sec. Last updated by anonymous at March 01 2017, 3:17:12 AM.

Summary

FINISHED

Spark SQL is a Spark library that makes it simple to do fast analysis of structured data. It provides more than just a SQL interface to Spark. It improves Spark usability, developer productivity, and application performance.

Spark SQL provides a unified interface for processing structured data from a variety of data sources. It can be used to process data stored on a local file system, HDFS, S3, JDBC-compliant databases, and NoSQL datastores. It allows you to process data from any of these data sources using SQL, HiveQL, or the DataFrame API.

Spark SQL also provides a distributed SQL query server that supports Thrift, JDBC and ODBC clients. It allows you to interactively analyze data using just SQL/HiveQL. The Spark SQL Thrift/JDBC/ODBC server can be used with any third-party BI or data visualization application that supports the JDBC or ODBC interface.

Took 0 sec. Last updated by anonymous at March 01 2017, 2:56:35 AM. (outdated)

```
%md
```

READY