Spark Tutorials

By Nadim Bahadoor (http://allaboutscala.com/nadim-bahadoor-founder-allaboutscala/) | Last updated: August 14, 2017 at 20:37 pm

In this section, we will show how to use Apache Spark (https://spark.apache.org/docs/2.0.2/) using **IntelliJ IDE** and **Scala**. The Apache Spark eco-system is moving at a fast pace and the tutorial will demonstrate the features of the latest **Apache Spark 2** version.

If you are not familiar with IntelliJ and Scala, feel free to review our previous tutorials on IntelliJ (http://allaboutscala.com/tutorials/chapter-1-getting-familiar-intellij-ide/) and Scala (http://allaboutscala.com/). So let's get started!

Source Code:

- The source code is available on the allaboutscala GitHub repository (https://github.com/nadimbahadoor).

Project Setup:

- StackOverflow dataset
- Add Apache Spark 2.0 SBT dependencies
- Bootstrap a SparkSession

DataFrame SQL Query:

- DataFrame Introduction
- Create a DataFrame from reading a CSV file
- DataFrame schema
- Select columns from a dataframe
- Filter by column value of a dataframe

- Count rows of a dataframe
- SQL like query
- Multiple filter chaining
- SQL IN clause
- SQL Group By
- SQL Group By with filter
- SQL order by
- Cast columns to specific data type
- Operate on a filtered dataframe
- DataFrame Join
- Join and select columns
- Join on explicit columns
- Inner Join
- Left Outer Join
- Right Outer Join
- Distinct

Spark SQL:

- Spark SQL Introduction
- Register temp table from dataframe
- List all tables in Spark's catalog
- List catalog tables using Spark SQL
- Select columns
- Filter by column value
- Count number of rows
- SQL like
- SQL where with and clause
- SQL IN clause
- SQL Group By
- SQL Group By with having clause
- SQL Order by
- Typed columns, filter and create temp table
- SQL Inner Join
- SQL Left Outer Join
- SQL Right Outer Join
- SQL Distinct
- Register User Defined Function (UDF)

DataFrame Statistics:

- DataFrame Statistics Introduction
- Create DataFrame from CSV
- Average
- Maximum
- Minimum
- Mean
- Sum
- Group by query with statistics
- DataFrame Statistics using describe() method
- Correlation
- Covariance
- Frequent Items
- Crosstab
- Stratified sampling using sampleBy

- Approximate Quantile
- Bloom Filter
- Count Min Sketch
- Sampling With Replacement

DataFrame Operations:

- DataFrame Operations Introduction
- Setup DataFrames
- Convert DataFrame row to Scala case class
- DataFrame row to Scala case class using map()
- Create DataFrame from collection
- DataFrame Union
- DataFrame Intersection
- Append column to DataFrame using withColumn()

More examples on the way ... stay tuned!

StackOverflow dataset

We will make use of the open-sourced StackOverflow dataset (https://github.com/dgrtwo/StackLite). We've cut down each dataset to just 10K line items for the purpose of showing how to use Apache Spark **DataFrame** and Apache Spark **SQL**.

The first dataset is called **question_tags_10K.csv** and it has the following data columns:

```
Id,Tag
1,data
4,c#
4,winforms
4,type-conversion
4,decimal
4,opacity
6,html
6,css
6,css3
```

The second dataset is called **questions_10K.csv** and it has the following data columns:

```
Id,CreationDate,ClosedDate,DeletionDate,Score,OwnerUserId,AnswerCount
1,2008-07-31T21:26:37Z,NA,2011-03-28T00:53:47Z,1,NA,0
4,2008-07-31T21:42:52Z,NA,NA,472,8,13
6,2008-07-31T22:08:08Z,NA,NA,210,9,5
8,2008-07-31T23:33:19Z,2013-06-03T04:00:25Z,2015-02-11T08:26:40Z,42,NA,8
9,2008-07-31T23:40:59Z,NA,NA,1452,1,58
11,2008-07-31T23:55:37Z,NA,NA,1154,1,33
13,2008-08-01T00:42:38Z,NA,NA,464,9,25
14,2008-08-01T00:59:11Z,NA,NA,296,11,8
16,2008-08-01T04:59:33Z,NA,NA,84,2,5
```

We will put both of these datasets under the **resources** directory - see GitHub source code (https://github.com/nadimbahadoor).

Add Apache Spark 2 SBT dependencies

In our **build.sbt** file, we need to tell SBT to import the Apache Spark 2 dependencies as shown below. You can learn more about importing SBT dependencies from this tutorial (http://allaboutscala.com/tutorials/chapter-1-getting-familiar-intellij-ide/intellij-import-dependencies-sbt-maven/).

```
name := "learn-spark"
```

```
version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies ++= Seq(
  "org.apache.spark"  %%  "spark-core"    % "2.2.0"   % "provided",
  "org.apache.spark"  %%  "spark-sql"     % "2.2.0",
  "org.apache.spark"  %%  "spark-mllib"   % "2.2.0"
)
```

Bootstrap a SparkSession

To connect to a Spark cluster, you need to create a **spark session** and we will encapsulate this behaviour into a simple **trait**. For more details on traits, refer to the Chapter on Scala Traits (http://allaboutscala.com/tutorials/chapter-5-traits/).

```
trait Context {

  lazy val sparkConf = new SparkConf()
    .setAppName("Learn Spark")
    .setMaster("local[*]")
    .set("spark.cores.max", "2")

  lazy val sparkSession = SparkSession
    .builder()
    .config(sparkConf)
    .getOrCreate()
}
```

NOTE:

- A **SparkSession** takes a **SparkConf** where we've specified a **name** for our Spark application, the Spark **master** which is our local node and also have limited the use of only 2 **cores**.
- For additional configuration properties for SparkConf, see the official Apache Spark documentation (https://spark.apache.org/docs/latest/).

DataFrame introduction

The examples in this section will make use of the **Context** trait which we've created in Bootstrap a SparkSession. By extending the Context trait, we will have access to a **SparkSession**.

```
object DataFrame_Tutorial extends App with Context {

}
```

Create a DataFrame from reading a CSV file

To create a DataFrame from reading a CSV file we will make use of the SparkSession and call the **read** method. Since the CSV file **question_tags_10K.csv** has two columns **id** and **tag**, we call the **toDF()** method.

To visually inspect some of the data points from our dataframe, we call the method **show(10)** which will print only 10 line items to the console.

```
  // Create a DataFrame from reading a CSV file
  val dfTags = sparkSession
    .read
```

```
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("src/main/resources/question_tags_10K.csv")
    .toDF("id", "tag")

  dfTags.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+------------------+
| id|               tag|
+---+------------------+
|  1|              data|
|  4|                c#|
|  4|           winforms|
|  4|    type-conversion|
|  4|           decimal|
|  4|           opacity|
|  6|              html|
|  6|               css|
|  6|              css3|
|  6|internet-explorer-7|
+---+------------------+
only showing top 10 rows
```

Print DataFrame schema

When creating the **dfTags** DataFrame, we specified the option to infer schema using: **option("inferSchema", "true")** This essentially instructs Spark to automatically infer the data type for each column when reading the CSV file **question_tags_10K.csv**

To show the dataframe schema which was inferred by Spark, you can call the method **printSchema()** on the dataframe **dfTags**.

```
  // Print DataFrame schema
  dfTags.printSchema()
```

You should see the following output when you run your Scala application in IntelliJ:

```
root
 |-- id: integer (nullable = true)
 |-- tag: string (nullable = true)
```

NOTE:

- Spark correctly inferred that the **id** column is of **integer** datatype and the **tag** column is of **string** type.

DataFrame Query: select columns from a dataframe

To select specific columns from a dataframe, you can use the **select()** method and pass in the columns which you want to select.

```
  // Query dataframe: select columns from a dataframe
  dfTags.select("id", "tag").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+------------------+
| id|               tag|
+---+------------------+
|  1|              data|
|  4|                c#|
|  4|           winforms|
|  4|    type-conversion|
```

```
|  4|           decimal|
|  4|           opacity|
|  6|              html|
|  6|               css|
|  6|              css3|
|  6|internet-explorer-7|
+---+-------------------+
only showing top 10 rows
```

DataFrame Query: filter by column value of a dataframe

To find all rows matching a specific column value, you can use the **filter()** method of a dataframe. For example, let's find all rows where the **tag** column has a value of **php**.

```
  // DataFrame Query: filter by column value of a dataframe
  dfTags.filter("tag == 'php'").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+---+
| id|tag|
+---+---+
| 23|php|
| 42|php|
| 85|php|
|126|php|
|146|php|
|227|php|
|249|php|
|328|php|
|588|php|
|657|php|
+---+---+
only showing top 10 rows
```

DataFrame Query: count rows of a dataframe

To count the number of rows in a dataframe, you can use the **count()** method. Note also that you can **chain** Spark DataFrame's method. As an example, let's **count** the number of **php tags** in our dataframe **dfTags**.

```
  // DataFrame Query: count rows of a dataframe
  println(s"Number of php tags = ${ dfTags.filter("tag == 'php'").count() }")
```

You should see the following output when you run your Scala application in IntelliJ:

```
Number of php tags = 133
```

NOTE:

- We've chained the **filter()** and **count()** methods.

DataFrame Query: SQL like query

We've already seen that you can query a dataframe column and find an exact value match using the **filter()** method. In addition to finding the exact value, you can also query a dataframe column's value using a familiar **SQL like** clause.

As an example, let us find all tags whose value start with the letter s.

```
  // DataFrame Query: SQL like query
  dfTags.filter("tag like 's%'").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+-------------+
| id|          tag|
+---+-------------+
| 25|      sockets|
| 36|          sql|
| 36|   sql-server|
| 40| structuremap|
| 48|submit-button|
| 79|          svn|
| 79|    subclipse|
| 85|          sql|
| 90|          svn|
|108|          svn|
+---+-------------+
only showing top 10 rows
```

DataFrame Query: Multiple filter chaining

From our previous examples, you should already be aware that Spark allows you to chain multiple dataframe operations. With that in mind, let us expand the previous example and add one more **filter()** method.

Our query below will find all tags whose value starts with letter s and then only pick id 25 or 108.

```
  // DataFrame Query: Multiple filter chaining
  dfTags
    .filter("tag like 's%'")
    .filter("id == 25 or id == 108")
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+-------+
| id|    tag|
+---+-------+
| 25|sockets|
|108|    svn|
+---+-------+
```

DataFrame Query: SQL IN clause

In the previous example, we saw how to slice our data using the **OR** clause to include only id 25 or 108 using **.filter("id == 25 or id == 108")**. Similarly, we can make use of a **SQL IN** clause to find all tags whose ids are equal to (25, 108).

```
  // DataFrame Query: SQL IN clause
  dfTags.filter("id in (25, 108)").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+---------+
| id|      tag|
+---+---------+
| 25|      c++|
| 25|        c|
| 25|  sockets|
| 25|mainframe|
| 25|      zos|
|108|  windows|
|108|      svn|
```

```
|108|    64bit|
+---+---------+
```

DataFrame Query: SQL Group By

We can use the **groupBy()** method on a dataframe to execute a similar SQL group by query. As an example, let us find out how many rows match each tag in our dataframe **dfTags**.

```
  // DataFrame Query: SQL Group By
  println("Group by tag value")
  dfTags.groupBy("tag").count().show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+--------------------+-----+
|                 tag|count|
+--------------------+-----+
|         type-safety|    4|
|             jbutton|    1|
|              iframe|    2|
|           svn-hooks|    2|
|           standards|    7|
|knowledge-management|    2|
|            trayicon|    1|
|           arguments|    1|
|                 zfs|    1|
|              import|    3|
+--------------------+-----+
only showing top 10 rows
```

DataFrame Query: SQL Group By with filter

We can further expand the previous group by example and only display tags that have more than 5 matching rows.

```
  // DataFrame Query: SQL Group By with filter
  dfTags.groupBy("tag").count().filter("count > 5").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------------+-----+
|             tag|count|
+----------------+-----+
|       standards|    7|
|        keyboard|    8|
|             rss|   12|
|   documentation|   15|
|         session|    6|
|build-automation|    9|
|            unix|   34|
|          iphone|   16|
|             xss|    6|
| database-design|   12|
+----------------+-----+
only showing top 10 rows
```

DataFrame Query: SQL order by

To complete the previous example which was a **group by** query along with a count, let us also sort the final results by adding an **order** by clause.

```
  // DataFrame Query: SQL order by
  dfTags.groupBy("tag").count().filter("count > 5").orderBy("tag").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------------+-----+
|             tag|count|
+----------------+-----+
|            .net|  351|
|        .net-2.0|   14|
|        .net-3.5|   30|
|           64bit|    7|
|  actionscript-3|   22|
|active-directory|   10|
|         ado.net|   11|
|           adobe|    7|
|           agile|    8|
|             air|   11|
+----------------+-----+
only showing top 10 rows
```

DataFrame Query: Cast columns to specific data type

So far, we've been using our dataframe **dfTags** which we read from StackOverflow's **question_tags_10K.csv** file. Let us now read the second StackOverflow file **questions_10K.csv** using a similar approach as we did for reading the tags file.

```
// DataFrame Query: Cast columns to specific data type
val dfQuestionsCSV = sparkSession
.read
.option("header", "true")
.option("inferSchema", "true")
.option("dateFormat","yyyy-MM-dd HH:mm:ss")
.csv("src/main/resources/questions_10K.csv")
.toDF("id", "creation_date", "closed_date", "deletion_date", "score", "owner_userid", "answer_count")

dfQuestionsCSV.printSchema()
```

You should see the following output when you run your Scala application in IntelliJ:

```
root
 |-- id: integer (nullable = true)
 |-- creation_date: timestamp (nullable = true)
 |-- closed_date: string (nullable = true)
 |-- deletion_date: string (nullable = true)
 |-- score: integer (nullable = true)
 |-- owner_userid: string (nullable = true)
 |-- answer_count: string (nullable = true)
```

NOTE:

- Although we've passed in the **inferSchema** option, Spark did not fully match the data type for some of our columns. Column **closed_date** is of type **string** and so is column **owner_userid** and **answer_count**.
- There are a few ways to be explicit about our column data types and for now we will show how to explicitly using the **cast** feature for the date fields.

```
val dfQuestions = dfQuestionsCSV.select(
dfQuestionsCSV.col("id").cast("integer"),
dfQuestionsCSV.col("creation_date").cast("timestamp"),
dfQuestionsCSV.col("closed_date").cast("timestamp"),
dfQuestionsCSV.col("deletion_date").cast("date"),
dfQuestionsCSV.col("score").cast("integer"),
dfQuestionsCSV.col("owner_userid").cast("integer"),
dfQuestionsCSV.col("answer_count").cast("integer")
```

```
)

dfQuestions.printSchema()
dfQuestions.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
root
 |-- id: integer (nullable = true)
 |-- creation_date: timestamp (nullable = true)
 |-- closed_date: timestamp (nullable = true)
 |-- deletion_date: date (nullable = true)
 |-- score: integer (nullable = true)
 |-- owner_userid: integer (nullable = true)
 |-- answer_count: integer (nullable = true)

+---+------------------+------------------+-------------+-----+------------+------------+
| id|      creation_date|       closed_date|deletion_date|score|owner_userid|answer_count|
+---+------------------+------------------+-------------+-----+------------+------------+
| Id|              null|              null|         null|Score| OwnerUserId| AnswerCount|
|  1|2008-07-31 22:26:37|              null|   2011-03-28|    1|          NA|           0|
|  4|2008-07-31 22:42:52|              null|         null|  472|           8|          13|
|  6|2008-07-31 23:08:08|              null|         null|  210|           9|           5|
|  8|2008-08-01 00:33:19|2013-06-03 05:00:25|   2015-02-11|   42|          NA|           8|
|  9|2008-08-01 00:40:59|              null|         null| 1452|           1|          58|
| 11|2008-08-01 00:55:37|              null|         null| 1154|           1|          33|
| 13|2008-08-01 01:42:38|              null|         null|  464|           9|          25|
| 14|2008-08-01 01:59:11|              null|         null|  296|          11|           8|
| 16|2008-08-01 05:59:33|              null|         null|   84|           2|           5|
+---+------------------+------------------+-------------+-----+------------+------------+
only showing top 10 rows
```

NOTE:

- All our columns for the questions dataframe now seem sensible with columns **id**, **score**, **owner_userid** and **answer_count** mapped to **integer** type, columns **creation_date** and **closed_date** are of type **timestamp** and **deletion_date** is of **type** date.

DataFrame Query: Operate on a filtered dataframe

The previous **dfQuestions** dataframe is great but perhaps we'd like to work with just a subset of the questions data. As an example, let us use our familiar **filter()** method to slice our dataframe **dfQuestions** with rows where the **score** is greater than **400** and less than **410**.

You should already be familiar with such filtering from the previous examples. However, if you look closely, you would notice that we can in fact assign the **filter()** operation to a **val** of type **dataframe!** We will use the **dfQuestionsSubset** dataframe to show how to execute **join** queries by joining it with the **dfTags** dataframe.

```
// DataFrame Query: Operate on a sliced dataframe
val dfQuestionsSubset = dfQuestions.filter("score > 400 and score < 410").toDF()
dfQuestionsSubset.show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+-----+------------------+------------------+-------------+-----+------------+------------+
|   id|      creation_date|       closed_date|deletion_date|score|owner_userid|answer_count|
+-----+------------------+------------------+-------------+-----+------------+------------+
|  888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
| 3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|
|16100|2008-08-19 13:51:55|              null|         null|  406|         203|          19|
|28098|2008-08-26 14:56:49|              null|         null|  403|        2680|          23|
|28637|2008-08-26 18:09:45|              null|         null|  401|        2469|          15|
|41479|2008-09-03 12:29:57|              null|         null|  406|        3394|          86|
```

```
|50467|2008-09-08 20:21:19|               null|         null|  402|        1967|          34|
|56628|2008-09-11 15:08:11|               null|         null|  403|        5469|          19|
|64860|2008-09-15 18:21:31|               null|         null|  402|        2948|          12|
+-----+-------------------+--------------------+-------------+-----+------------+------------+
```

DataFrame Query: Join

We will now make use of the previous **dfQuestionsSubset** dataframe. In this example, we will **join** the dataframe **dfQuestionsSubset** with the tags dataframe **dfTags** by the **id** column.

```
  // DataFrame Query: Join
  dfQuestionsSubset.join(dfTags, "id").show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
|  id|      creation_date|       closed_date|deletion_date|score|owner_userid|answer_count|                 tag|
+----+-------------------+------------------+-------------+-----+------------+------------+--------------------+
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|              xdebug|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|            phpstorm|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|           debugging|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|             eclipse|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|                 php|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 osx|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 ios|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|         objective-c|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|              iphone|
|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|illegalargumentex...|
+----+-------------------+------------------+-------------+-----+------------+------------+--------------------+
only showing top 10 rows
```

DataFrame Query: Join and select columns

To follow up on the previous example, you can chain the **select()** method after the **join()** in order to only display certain columns.

```
  // DataFrame Query: Join and select columns
  dfQuestionsSubset
    .join(dfTags, "id")
    .select("owner_userid", "tag", "creation_date", "score")
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
|owner_userid|                 tag|      creation_date|score|
+------------+--------------------+-------------------+-----+
|         131|              xdebug|2008-08-04 00:18:21|  405|
|         131|            phpstorm|2008-08-04 00:18:21|  405|
|         131|           debugging|2008-08-04 00:18:21|  405|
|         131|             eclipse|2008-08-04 00:18:21|  405|
|         131|                 php|2008-08-04 00:18:21|  405|
|        null|                 osx|2008-08-05 06:39:36|  408|
|        null|                 ios|2008-08-05 06:39:36|  408|
|        null|         objective-c|2008-08-05 06:39:36|  408|
|        null|              iphone|2008-08-05 06:39:36|  408|
|         122|illegalargumentex...|2008-08-06 20:26:30|  402|
+------------+--------------------+-------------------+-----+
only showing top 10 rows
```

DataFrame Query: Join on explicit columns

A **join()** operation will join two dataframes based on some common column which in the previous example was the column **id** from **dfTags** and **dfQuestionsSubset**. But, what if the column to join to had different names? In such a case, you can explicitly specify the column from each dataframe on which to join.

```
// DataFrame Query: Join on explicit columns
dfQuestionsSubset
  .join(dfTags, dfTags("id") === dfQuestionsSubset("id"))
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+-------------------+-------------------+-------------+-----+------------+------------+----+--------------------+
|  id|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|  id|                 tag|
+----+-------------------+-------------------+-------------+-----+------------+------------+----+--------------------+
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30| 888|              xdebug|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30| 888|            phpstorm|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30| 888|           debugging|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30| 888|             eclipse|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30| 888|                 php|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|1939|                 osx|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|1939|                 ios|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|1939|         objective-c|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|1939|              iphone|
|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|3881|illegalargumentex...|
+----+-------------------+-------------------+-------------+-----+------------+------------+----+--------------------+
only showing top 10 rows
```

DataFrame Query: Inner Join

Spark supports various types of joins namely: **inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, left_anti**. These are specified in the official Apache Spark Documentation (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset). In this example, we will show how to use the **inner join** type.

```
// DataFrame Query: Inner Join
dfQuestionsSubset
  .join(dfTags, Seq("id"), "inner")
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
|  id|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|                 tag|
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|              xdebug|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|            phpstorm|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|           debugging|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|             eclipse|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|                 php|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 osx|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 ios|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|         objective-c|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|              iphone|
|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|illegalargumentex...|
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
only showing top 10 rows
```

DataFrame Query: Left Outer Join

Following on from the previous **inner join** example, the code below shows how to perform a **left outer join** in Apache Spark.

```
// DataFrame Query: Left Outer Join
dfQuestionsSubset
  .join(dfTags, Seq("id"), "left_outer")
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
|  id|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|                 tag|
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|              xdebug|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|            phpstorm|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|           debugging|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|             eclipse|
| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|                 php|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 osx|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|                 ios|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|         objective-c|
|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|              iphone|
|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|illegalargumentex...|
+----+-------------------+-------------------+-------------+-----+------------+------------+--------------------+
only showing top 10 rows
```

DataFrame Query: Right Outer Join

As mentioned in the previous join examples, Apache Spark supports a number of join types as listed in the official Apache Spark Documentation (https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset). Let's show one more join type which is the **right outer join**. Note that we've swapped the dataframes ordering for the **right outer join** by joining **dfTags** with **dfQuestionsSubset**.

```
// DataFrame Query: Right Outer Join
dfTags
  .join(dfQuestionsSubset, Seq("id"), "right_outer")
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+--------------------+-------------------+-------------------+-------------+-----+------------+------------+
|  id|                 tag|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|
+----+--------------------+-------------------+-------------------+-------------+-----+------------+------------+
| 888|              xdebug|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|            phpstorm|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|           debugging|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|             eclipse|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|                 php|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
|1939|                 osx|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|                 ios|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|         objective-c|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|              iphone|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|3881|illegalargumentex...|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|
+----+--------------------+-------------------+-------------------+-------------+-----+------------+------------+
only showing top 10 rows
```

DataFrame Query: Distinct

With the dataframe dfTags in scope, we can find the unique values in the tag column by using the **distinct()** method.

```
// DataFrame Query: Distinct
dfTags
  .select("tag")
  .distinct()
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+--------------------+
|                 tag|
+--------------------+
|         type-safety|
|             jbutton|
|              iframe|
|           svn-hooks|
|           standards|
|knowledge-management|
|            trayicon|
|           arguments|
|                 zfs|
|              import|
+--------------------+
only showing top 10 rows
```

Spark SQL Introduction

In this section, we will show how to use **Apache Spark SQL** which brings you much closer to an **SQL** style query similar to using a relational database. We will once more reuse the Context trait which we created in Bootstrap a SparkSession so that we can have access to a **SparkSession**.

```
object SparkSQL_Tutorial extends App with Context {

}
```

Register temp table from dataframe

By now you should be familiar with how to create a dataframe from reading a csv file. The code below will first create a dataframe for the StackOverflow **question_tags_10K.csv** file which we will name **dfTags**. But instead of operating directly on the dataframe **dfTags**, we will register it as a **temporary table** in Spark's **catalog** and name the table **so_tags**.

```
  // Register temp table
  val dfTags = sparkSession
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("src/main/resources/question_tags_10K.csv")
    .toDF("id", "tag")

  dfTags.createOrReplaceTempView("so_tags")
```

List all tables in Spark's catalog

To verify that the temporary table **so_tags** has been in fact registered into Spark's catalog, you can access and call **catalog** related methods on the SparkSession as follows:

```
  // List all tables in Spark's catalog
  sparkSession.catalog.listTables().show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+-------+--------+-----------+---------+-----------+
|   name|database|description|tableType|isTemporary|
+-------+--------+-----------+---------+-----------+
|so_tags|    null|       null|TEMPORARY|       true|
+-------+--------+-----------+---------+-----------+
```

List catalog tables using Spark SQL

To issue SQL like queries, you can make use of the **sql()** method on the SparkSession and pass in a query string. Let's redo the previous example and list all tables in Spark's **catalog** using **Spark SQL** query.

```
// List all tables in Spark's catalog using Spark SQL
sparkSession.sql("show tables").show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+--------+---------+-----------+
|database|tableName|isTemporary|
+--------+---------+-----------+
|        |  so_tags|       true|
+--------+---------+-----------+
```

Select columns

In the previous example, we showed how to use the **sql()** method to issue **SQL** style queries. We will now re-write the dataframe queries using **Spark SQL**.

```
// Select columns
sparkSession
  .sql("select id, tag from so_tags limit 10")
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+------------------+
| id|               tag|
+---+------------------+
|  1|              data|
|  4|                c#|
|  4|          winforms|
|  4|   type-conversion|
|  4|           decimal|
|  4|           opacity|
|  6|              html|
|  6|               css|
|  6|              css3|
|  6|internet-explorer-7|
+---+------------------+
```

NOTE:

- This Spark SQL query is similar to the dataframe select columns example.

Filter by column value

In the DataFrame SQL query, we showed how to filter a dataframe by a column value. We can re-write the example using **Spark SQL** as shown below.

```
// Filter by column value
sparkSession
  .sql("select * from so_tags where tag = 'php'")
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+---+
| id|tag|
```

```
+---+---+
| 23|php|
| 42|php|
| 85|php|
|126|php|
|146|php|
|227|php|
|249|php|
|328|php|
|588|php|
|657|php|
+---+---+
only showing top 10 rows
```

Count number of rows

In the DataFrame SQL query, we showed how to count rows of a dataframe. We can re-write the count number of php tags example using **Spark SQL** as shown below.

```
  // Count number of rows
  sparkSession
    .sql(
      """select
              |count(*) as php_count
              |from so_tags where tag='php'""".stripMargin)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---------+
|php_count|
+---------+
|      133|
+---------+
```

SQL like

In the DataFrame SQL query, we showed how to issue SQL like query. We can re-write the dataframe like query to find all tags which start with the letter s using **Spark SQL** as shown below.

```
  // SQL like
  sparkSession
    .sql(
      """select *
              |from so_tags
              |where tag like 's%'""".stripMargin)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+-------------+
| id|          tag|
+---+-------------+
| 25|      sockets|
| 36|          sql|
| 36|   sql-server|
| 40| structuremap|
| 48|submit-button|
| 79|          svn|
| 79|    subclipse|
| 85|          sql|
| 90|          svn|
```

```
|108|          svn|
+---+-------------+
only showing top 10 rows
```

SQL where with and clause

In the DataFrame SQL query, we showed how to chain multiple filters on a dataframe. We can re-write the dataframe filter for tags starting the letter s and whose id is either 25 or 108 using **Spark SQL** as shown below.

```
// SQL where with and clause
sparkSession
  .sql(
    """select *
            |from so_tags
            |where tag like 's%'
            |and (id = 25 or id = 108)""".stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+-------+
| id|    tag|
+---+-------+
| 25|sockets|
|108|    svn|
+---+-------+
```

SQL IN clause

In the DataFrame SQL query, we showed how to issue an SQL in clause on a dataframe. We can re-write the dataframe in query to find tags whose id are in (25, 108) using **Spark SQL** as shown below.

```
// SQL IN clause
sparkSession
  .sql(
    """select *
            |from so_tags
            |where id in (25, 108)""".stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+---------+
| id|      tag|
+---+---------+
| 25|      c++|
| 25|        c|
| 25|  sockets|
| 25|mainframe|
| 25|      zos|
|108|  windows|
|108|      svn|
|108|    64bit|
+---+---------+
```

SQL Group By

In the DataFrame SQL query, we showed how to issue an SQL group by query on a dataframe. We can re-write the dataframe group by tag and count query using **Spark SQL** as shown below.

```
// SQL Group By
```

```
  sparkSession
    .sql(
      """select tag, count(*) as count
              |from so_tags group by tag""".stripMargin)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+--------------------+-----+
|                 tag|count|
+--------------------+-----+
|         type-safety|    4|
|             jbutton|    1|
|              iframe|    2|
|           svn-hooks|    2|
|           standards|    7|
|knowledge-management|    2|
|            trayicon|    1|
|           arguments|    1|
|                 zfs|    1|
|              import|    3|
+--------------------+-----+
only showing top 10 rows
```

SQL Group By with having clause

In the DataFrame SQL query, we showed how to issue an SQL group by with filter query on a dataframe. We can re-write the dataframe group by tag and count where count is greater than 5 query using **Spark SQL** as shown below.

```
  // SQL Group By with having clause
  sparkSession
    .sql(
      """select tag, count(*) as count
              |from so_tags group by tag having count > 5""".stripMargin)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------------+-----+
|             tag|count|
+----------------+-----+
|       standards|    7|
|        keyboard|    8|
|             rss|   12|
|   documentation|   15|
|         session|    6|
|build-automation|    9|
|            unix|   34|
|          iphone|   16|
|             xss|    6|
| database-design|   12|
+----------------+-----+
only showing top 10 rows
```

SQL Order by

In the DataFrame SQL query, we showed how to issue an SQL order by query on a dataframe. We can re-write the dataframe group by, count and order by tag query using **Spark SQL** as shown below.

```
  // SQL Order by
  sparkSession
    .sql(
```

```
      """select tag, count(*) as count
              |from so_tags group by tag having count > 5 order by tag""".stripMargin)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------------+-----+
|             tag|count|
+----------------+-----+
|            .net|  351|
|        .net-2.0|   14|
|        .net-3.5|   30|
|           64bit|    7|
|  actionscript-3|   22|
|active-directory|   10|
|         ado.net|   11|
|           adobe|    7|
|           agile|    8|
|             air|   11|
+----------------+-----+
only showing top 10 rows
```

Typed columns, filter and create temp table

In the DataFrame SQL query, we showed how to cast columns to specific data types and how to filter dataframe. We will use these examples to register a **temporary** table named **so_questions** for the StackOverflow's questions file: questions_10K.csv. The **so_questions** and **so_tags** tables will later be used to show how to do **SQL joins**.

```
  // Typed dataframe, filter and temp table
  val dfQuestionsCSV = sparkSession
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .option("dateFormat","yyyy-MM-dd HH:mm:ss")
    .csv("src/main/resources/questions_10K.csv")
    .toDF("id", "creation_date", "closed_date", "deletion_date", "score", "owner_userid", "answer_count")


  // cast columns to data types
  val dfQuestions = dfQuestionsCSV.select(
    dfQuestionsCSV.col("id").cast("integer"),
    dfQuestionsCSV.col("creation_date").cast("timestamp"),
    dfQuestionsCSV.col("closed_date").cast("timestamp"),
    dfQuestionsCSV.col("deletion_date").cast("date"),
    dfQuestionsCSV.col("score").cast("integer"),
    dfQuestionsCSV.col("owner_userid").cast("integer"),
    dfQuestionsCSV.col("answer_count").cast("integer")
  )

  // filter dataframe
  val dfQuestionsSubset = dfQuestions.filter("score > 400 and score < 410").toDF()

  // register temp table
  dfQuestionsSubset.createOrReplaceTempView("so_questions")
```

SQL Inner Join

In the DataFrame SQL query, we showed how to issue an SQL inner join on two dataframes. We can re-write the dataframe tags **inner join** with the dataframe questions using **Spark SQL** as shown below. Note also that we are using the two temporary tables which we created earlier namely **so_tags** and **so_questions**.

```
// SQL Inner Join
sparkSession
  .sql(
  """select t.*, q.*
        |from so_questions q
        |inner join so_tags t
        |on t.id = q.id"""   .stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
|  id|                 tag|  id|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
| 888|              xdebug| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|            phpstorm| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|           debugging| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|             eclipse| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|                 php| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
|1939|                 osx|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|                 ios|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|         objective-c|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|              iphone|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|3881|illegalargumentex...|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
only showing top 10 rows
```

SQL Left Outer Join

In the DataFrame SQL query, we showed how to issue an SQL left outer join on two dataframes. We can re-write the dataframe tags **left outer join** with the dataframe questions using **Spark SQL** as shown below. Note also that we are using the two temporary tables which we created earlier namely **so_tags** and **so_questions**.

```
// SQL Left Outer Join
sparkSession
  .sql(
    """select t.*, q.*
           |from so_questions q
           |left outer join so_tags t
           |on t.id = q.id"""        .stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
|  id|                 tag|  id|      creation_date|        closed_date|deletion_date|score|owner_userid|answer_count|
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
| 888|              xdebug| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|            phpstorm| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|           debugging| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|             eclipse| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|                 php| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
|1939|                 osx|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|                 ios|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|         objective-c|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|              iphone|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|3881|illegalargumentex...|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|
+----+--------------------+----+-------------------+-------------------+-------------+-----+------------+------------+
only showing top 10 rows
```

SQL Right Outer Join

In the DataFrame SQL query, we showed how to issue an SQL right outer join on two dataframes. We can re-write the dataframe tags **right outer join** with the dataframe questions using **Spark SQL** as shown below. Note also that we are using the two temporary tables which we created earlier namely **so_tags** and **so_questions**.

```
// SQL Right Outer Join
sparkSession
  .sql(
    """select t.*, q.*
            |from so_tags t
            |right outer join so_questions q
            |on t.id = q.id"""              .stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----+--------------------+----+--------------------+--------------------+-------------+-----+------------+------------+
|  id|                 tag|  id|      creation_date|       closed_date|deletion_date|score|owner_userid|answer_count|
+----+--------------------+----+--------------------+--------------------+-------------+-----+------------+------------+
| 888|              xdebug| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|            phpstorm| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|           debugging| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|             eclipse| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
| 888|                 php| 888|2008-08-04 00:18:21|2016-08-04 10:22:00|         null|  405|         131|          30|
|1939|                 osx|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|                 ios|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|         objective-c|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|1939|              iphone|1939|2008-08-05 06:39:36|2012-06-05 14:13:38|   2012-12-18|  408|        null|          48|
|3881|illegalargumentex...|3881|2008-08-06 20:26:30|2016-09-23 14:34:31|         null|  402|         122|          27|
+----+--------------------+----+--------------------+--------------------+-------------+-----+------------+------------+
only showing top 10 rows
```

SQL Distinct

In the DataFrame SQL query, we showed how to issue an SQL distinct on dataframe dfTags to find unique values in the tag column. We can re-write the dataframe tags **distinct** example using **Spark SQL** as shown below.

```
// SQL Distinct
sparkSession
  .sql("""select distinct tag from so_tags""".stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+--------------------+
|                 tag|
+--------------------+
|         type-safety|
|             jbutton|
|              iframe|
|           svn-hooks|
|           standards|
|knowledge-management|
|            trayicon|
|           arguments|
|                zfs|
|              import|
+--------------------+
only showing top 10 rows
```

Register User Defined Function (UDF)

For this example, we will show how **Apache Spark** allows you to register and use your own functions which are more commonly referred to as **User Defined Functions (UDF)**.

We will create a function named **prefixStackoverflow()** which will prefix the String value **so_** to a given String. In turn, we will register this function within our **Spark session** as a **UDF** and then use it in our **Spark SQL** query to augment each tag value with the prefix **so_**

```
// Function to prefix a String with so_ short for StackOverflow
def prefixStackoverflow(s: String): String = s"so_$s"

// Register User Defined Function (UDF)
sparkSession
  .udf
  .register("prefix_so", prefixStackoverflow _)

// Use udf prefix_so to augment each tag value with so_
sparkSession
  .sql("""select id, prefix_so(tag) from so_tags""".stripMargin)
  .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+--------------------+
| id|  UDF:prefix_so(tag)|
+---+--------------------+
|  1|             so_data|
|  4|               so_c#|
|  4|         so_winforms|
|  4|  so_type-conversion|
|  4|          so_decimal|
|  4|          so_opacity|
|  6|             so_html|
|  6|              so_css|
|  6|             so_css3|
|  6|so_internet-explo...|
+---+--------------------+
only showing top 10 rows
```

NOTE:

- Every tag value now has a prefix **so_**

DataFrame Statistics Introduction

The examples in this section will make use of the **Context** trait which we've created in Bootstrap a SparkSession. By extending the Context trait, we will have access to a **SparkSession**.

```
object DataFrameStatistics_Tutorial extends App {

}
```

Create DataFrame from CSV

To recap from the example on creating a dataframe from reading a CSV file, we will once again create two dataframes: one to the tags while the other for the questions StackOverflow CSV file.

```
// Create a dataframe from tags file question_tags_10K.csv
val dfTags = sparkSession
```

```
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("src/main/resources/question_tags_10K.csv")
    .toDF("id", "tag")

  // Create a dataframe from questions file questions_10K.csv
  val dfQuestionsCSV = sparkSession
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .option("dateFormat","yyyy-MM-dd HH:mm:ss")
    .csv("src/main/resources/questions_10K.csv")
    .toDF("id", "creation_date", "closed_date", "deletion_date", "score", "owner_userid", "answer_count")

  // cast columns to data types
  val dfQuestions = dfQuestionsCSV.select(
    dfQuestionsCSV.col("id").cast("integer"),
    dfQuestionsCSV.col("creation_date").cast("timestamp"),
    dfQuestionsCSV.col("closed_date").cast("timestamp"),
    dfQuestionsCSV.col("deletion_date").cast("date"),
    dfQuestionsCSV.col("score").cast("integer"),
    dfQuestionsCSV.col("owner_userid").cast("integer"),
    dfQuestionsCSV.col("answer_count").cast("integer")
  )
```

Average

With dataframe **dfQuestions** in scope, we will compute the **average** of the **score** column using the code below. Note that you also need to import Spark's built-in functions using: **import org.apache.spark.sql.functions._**

```
  // Average
  import org.apache.spark.sql.functions._
  dfQuestions
    .select(avg("score"))
    .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+-----------------+
|       avg(score)|
+-----------------+
|36.14631463146315|
+-----------------+
```

Maximum

With dataframe **dfQuestions** in scope, we will compute the **maximum** of the **score** column using the code below. Note that you also need to import Spark's built-in functions using: **import org.apache.spark.sql.functions._**

```
  // Max
  import org.apache.spark.sql.functions._
  dfQuestions
    .select(max("score"))
    .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------+
|max(score)|
+----------+
```

```
|      4443|
+----------+
```

Minimum

With dataframe **dfQuestions** in scope, we will compute the **minimum** of the **score** column using the code below. Note that you also need to import Spark's built-in functions using: **import org.apache.spark.sql.functions._**

```
// Minimum
import org.apache.spark.sql.functions._
dfQuestions
  .select(min("score"))
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------+
|min(score)|
+----------+
|       -27|
+----------+
```

Mean

With dataframe **dfQuestions** in scope, we will compute the **mean** of the **score** column using the code below. Note that you also need to import Spark's built-in functions using: **import org.apache.spark.sql.functions._**

```
// Mean
import org.apache.spark.sql.functions._
dfQuestions
  .select(mean("score"))
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+-----------------+
|       avg(score)|
+-----------------+
|36.14631463146315|
+-----------------+
```

Sum

With dataframe **dfQuestions** in scope, we will compute the **sum** of the **score** column using the code below. Note that you also need to import Spark's built-in functions using: **import org.apache.spark.sql.functions._**

```
// Sum
import org.apache.spark.sql.functions._
dfQuestions
  .select(sum("score"))
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------+
|sum(score)|
+----------+
|    361427|
+----------+
```

Group by with statistics

With the dataframe **dfQuestions** and **dfTags** in scope, we will apply what we've learned on DataFrame Query and DataFrame Statistics. The example below will find all questions where **id > 400** and **id < 450**, **filter** out any null in column owner_userid**, join** with **dfTags** on the id column, **group by** owner_userid and calculate the **average** score column and the **minimum** answer_count column.

```
// Group by with statistics
import org.apache.spark.sql.functions._
dfQuestions
  .filter("id > 400 and id < 450")
  .filter("owner_userid is not null")
  .join(dfTags, dfQuestions.col("id").equalTo(dfTags("id")))
  .groupBy(dfQuestions.col("owner_userid"))
  .agg(avg("score"), max("answer_count"))
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+------------+----------+-----------------+
|owner_userid|avg(score)|max(answer_count)|
+------------+----------+-----------------+
|         268|      26.0|                1|
|         136|      57.6|                9|
|         123|      20.0|                3|
+------------+----------+-----------------+
```

DataFrame Statistics using describe() method

In the previous examples, we've shown how to compute statistics on DataFrame. If you are looking for a quick shortcut to compute the **count**, **mean**, **standard deviation**, **min** and **max** values from a DataFrame, then you can use the **describe()** method as shown below:

```
// DataFrame Statistics using describe() method
val dfQuestionsStatistics = dfQuestions.describe()
dfQuestionsStatistics.show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+-------+-----------------+------------------+-----------------+------------------+
|summary|               id|             score|     owner_userid|      answer_count|
+-------+-----------------+------------------+-----------------+------------------+
|  count|             9999|              9999|             7388|              9922|
|   mean|33929.17081708171| 36.14631463146315|47389.99472116947|6.6232614392259626|
| stddev|19110.09560532429|160.48316753972045|280943.1070344427| 9.069109116851138|
|    min|                1|               -27|                1|                -5|
|    max|            66037|              4443|          3431280|               316|
+-------+-----------------+------------------+-----------------+------------------+
```

Correlation

For more advanced statistics which you typically add in a data science pipeline, Spark provides a convenient **stat** function. As an example, we will access the **correlation** method to find the **correlation** between column score and answer_count. For additional dataframe stat functions, see the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

```
// Correlation
val correlation = dfQuestions.stat.corr("score", "answer_count")
println(s"correlation between column score and answer_count = $correlation")
```

You should see the following output when you run your Scala application in IntelliJ:

```
correlation between column score and answer_count = 0.3699847903294707
```

Covariance

For more advanced statistics which you typically add in a data science pipeline, Spark provides a convenient **stat** function. As an example, we will access the **covariance** method to find the **covariance** between column score and answer_count. For additional dataframe stat functions, see the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

```
// Covariance
val covariance = dfQuestions.stat.cov("score", "answer_count")
println(s"covariance between column score and answer_count = $covariance")
```

You should see the following output when you run your Scala application in IntelliJ:

```
covariance between column score and answer_count = 537.513381444165
```

Frequent Items

For more advanced statistics which you typically add in a data science pipeline, Spark provides a convenient **stat** function. As an example, we will access the **freqItems** method to find the **frequent items** in the answer_count dataframe column. For additional dataframe stat functions, see the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

```
// Frequent Items
val dfFrequentScore = dfQuestions.stat.freqItems(Seq("answer_count"))
dfFrequentScore.show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------------------+
|answer_count_freqItems|
+----------------------+
|  [23, 131, 77, 86,...|
+----------------------+
```

Crosstab

For more advanced statistics which you typically add in a **data science** pipeline, Spark provides a convenient **stat** function. As an example, we will access the **crosstab** method to display a tabular view of score by owner_userid. For additional dataframe stat functions, see the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

```
// Crosstab
val dfScoreByUserid = dfQuestions
  .filter("owner_userid > 0 and owner_userid < 20")
  .stat
  .crosstab("score", "owner_userid")
dfScoreByUserid.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+------------------+---+---+---+---+---+---+---+---+---+---+
|score_owner_userid|  1| 11| 13| 17|  2|  3|  4|  5|  8|  9|
+------------------+---+---+---+---+---+---+---+---+---+---+
|                56|  0|  0|  0|  1|  0|  0|  0|  0|  0|  0|
|               472|  0|  0|  0|  0|  0|  0|  0|  0|  1|  0|
|                14|  0|  0|  0|  1|  0|  0|  0|  1|  0|  0|
|                20|  0|  0|  0|  0|  0|  0|  0|  1|  0|  0|
|               179|  0|  0|  0|  0|  0|  1|  0|  0|  0|  0|
|                84|  0|  0|  0|  0|  1|  0|  0|  0|  0|  0|
|               160|  0|  0|  1|  0|  0|  0|  0|  0|  0|  0|
|                21|  0|  0|  0|  0|  0|  0|  0|  1|  0|  0|
|                 9|  0|  0|  0|  0|  0|  0|  1|  1|  0|  0|
|                 2|  0|  0|  0|  0|  0|  0|  0|  1|  0|  1|
```

```
+------------------+---+---+---+---+---+---+---+---+---+---+
only showing top 10 rows
```

Stratified sampling using sampleBy

For more advanced statistics which you typically add in a **data science** pipeline, Spark provides a convenient **stat** function. In this section, we will show how to perform stratified sampling (https://en.wikipedia.org/wiki/Stratified_sampling) on a dataframe using the **sampleBy()** method. For additional dataframe stat functions, see the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

To start with, we will filter the dataframe **dfQuestions** to only include rows where answer_count is in (5, 10, 20). We then print the number of rows matching each answer_count so that we get an initial visual representation of the new dataframe **dfQuestionsByAnswerCount**.

```
// find all rows where answer_count in (5, 10, 20)
val dfQuestionsByAnswerCount = dfQuestions
  .filter("owner_userid > 0")
  .filter("answer_count in (5, 10, 20)")

// count how many rows match answer_count in (5, 10, 20)
dfQuestionsByAnswerCount
  .groupBy("answer_count")
  .count()
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+------------+-----+
|answer_count|count|
+------------+-----+
|          20|   34|
|           5|  811|
|          10|  272|
+------------+-----+
```

Next, we create a fraction map which is a Map of **key** and **values**. Key in our example is the answer_count: 5, 10, 20 and values are fractions of the number of rows which we are interested to sample. The **values** should be in the **range [0, 1]**. Below the fractions map implies that we are interested in:

- 50% of the rows that have answer_count = 5
- 10% of the rows that have answer_count = 10
- 100% of the rows that have answer_count = 20

With the fractions map defined, we need to pass it as a parameter to the **sampleBy()** method. Note also that you need to specify a random **seed** parameter as well.

```
// Create a fraction map where we are only interested:
// - 50% of the rows that have answer_count = 5
// - 10% of the rows that have answer_count = 10
// - 100% of the rows that have answer_count = 20
// Note also that fractions should be in the range [0, 1]
val fractionKeyMap = Map(5 -> 0.5, 10 -> 0.1, 20 -> 1.0)

// Stratified sample using the fractionKeyMap.
dfQuestionsByAnswerCount
  .stat
  .sampleBy("answer_count", fractionKeyMap, 7L)
  .groupBy("answer_count")
  .count()
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+------------+-----+
|answer_count|count|
+------------+-----+
|          20|   34|
|           5|  400|
|          10|   26|
+------------+-----+
```

Note that changing the random seed will modify your sampling outcome. As an example, let's change the random **seed** to **37**.

```
// Note that changing the random seed will modify your sampling outcome. As an example, let's change the random seed
dfQuestionsByAnswerCount
  .stat
  .sampleBy("answer_count", fractionKeyMap, 37L)
  .groupBy("answer_count")
  .count()
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+------------+-----+
|answer_count|count|
+------------+-----+
|          20|   34|
|           5|  388|
|          10|   25|
+------------+-----+
```

Approximate Quantile

For more advanced statistics which you typically add in a **data science** pipeline, Spark provides a convenient **stat** function. For instance, when doing data exploration, you sometimes want to find out **summary** about various **quantiles** in your dataset. Spark comes with a handy **approxQuantile()** method and more details about the internal implementations and other stat functions can be found on the official Spark 2 API documentation (https://spark.apache.org/docs/2.0.2/api/java/org/apache/spark/sql/DataFrameStatFunctions.html).

The first parameter of the **approxQuantile()** method is the **column** of your dataframe on which to run the statistics, the second parameter is an Array of **quantile probabilities** and the third parameter is a **precision error factor**.

In the example below, we will find the minimum, median and maximum from the score column and as such we will pass an Array of probabilities Array(0, 0.5, 1) which represents:

- 0 = minimum
- 0.5 = median
- 1 = maximum

```
// Approximate Quantile
val quantiles = dfQuestions
  .stat
  .approxQuantile("score", Array(0, 0.5, 1), 0.25)
println(s"Qauntiles segments = ${quantiles.toSeq}")
```

You should see the following output when you run your Scala application in IntelliJ:

```
Qauntiles segments = WrappedArray(-27.0, 2.0, 4443.0)
```

You can also verify the quantiles statistics above using **Spark SQL** as follows:

```
dfQuestions.createOrReplaceTempView("so_questions")
sparkSession
  .sql("select min(score), percentile_approx(score, 0.25), max(score) from so_questions")
  .show()
```

You should see the following output when you run your Scala application in IntelliJ:

```
+----------+-----------------------------------------------------------------------+----------+
|min(score)|percentile_approx(CAST(score AS DOUBLE), CAST(0.25 AS DOUBLE), 10000)|max(score)|
+----------+-----------------------------------------------------------------------+----------+
|       -27|                                                                    2.0|      4443|
+----------+-----------------------------------------------------------------------+----------+
```

Bloom Filter

For more advanced statistics which you typically add in a **data science** pipeline, Spark provides a convenient **stat** function. For instance, when training large datasets in a **Machine Learning** pipeline, you can make use of pre-processing steps such as Bloom Filtering (https://en.wikipedia.org/wiki/Bloom_filter) to compact storage requirements of intermediate steps and also improve performance of iterative algorithms. Spark comes with a handy **bloom filter** implementation and it is exposed under the **stat** function.

The first parameter of the **bloomFilter()** method is the **column** of your dataframe on which a bloom filter set will be created, the second parameter is the **number of items** in the bloom filter set and the third parameter is a **false positive factor (https://en.wikipedia.org/wiki/Type_I_and_type_II_errors)**.

In the example below, we will create a bloom filter for the tags column with **1000** items and a **10%** false positive factor. Note that **dfTags** has **10000** tags using DataFrame count (http://allaboutscala.com/big-data/spark/#dataframe-count-rows), but we are only storing **1000** items, i.e. 10% of the total number of tags, in the bloom filter below.

```
// Bloom Filter
val tagsBloomFilter = dfTags.stat.bloomFilter("tag", 1000L, 0.1)
```

Instead of querying the dataframe dfTags directly, we will use the **mightContain()** method of the Bloom Filter **tagsBloomFilter** to test whether certain tags exists.

```
println(s"bloom filter contains java tag = ${tagsBloomFilter.mightContain("java")}")
println(s"bloom filter contains some unknown tag = ${tagsBloomFilter.mightContain("unknown tag")}")
```

You should see the following output when you run your Scala application in IntelliJ:

```
bloom filter contains java tag = true
bloom filter contains some unknown tag = false
```

Count Min Sketch

For more advanced statistics which you typically add in a **data science** pipeline, Spark provides a convenient stat function. For instance, Spark supports the Count Min Sketch data structure (https://en.wikipedia.org/wiki/Count%E2%80%93min_sketch) typically used in probability approximation. The **countMinSketch()** method is exposed under the **stat** function.

As an example, we will create a **Count Min Sketch** data structure over the tag column of dataframe **dfTags** and estimate the occurrence for the term java.

The **countMinSketch()** method first parameter is the **column** of your dataframe to create the Count Min Sketch data structure, the second parameter is a **precision error factor**, the third parameter is the **confidence level** and the fourth parameter is a **random seed**.

In the example below, the various **Count Min Sketch** parameters are as follows:

- first parameter = the tag column of dataframe dfTags
- second parameter = 10% precision error factor
- third parameter = 90% confidence level
- fourth parameter = 37 as a random seed

```
// Count Min Sketch
val cmsTag = dfTags.stat.countMinSketch("tag", 0.1, 0.9, 37)
val estimatedFrequency = cmsTag.estimateCount("java")
println(s"Estimated frequency for tag java = $estimatedFrequency")
```

You should see the following output when you run your Scala application in IntelliJ:

```
Estimated frequency for tag java = 513
```

Sampling With Replacement

In the previous example above, we showed how to use Spark for Stratified Sampling using the sampleBy() method (http://allaboutscala.com/big-data/spark/#dataframe-statistics-sampleby). To support additional data analysis such as correlation (http://allaboutscala.com/big-data/spark/#dataframe-statistics-correlation) and covariance (http://allaboutscala.com/big-data/spark/#dataframe-statistics-covariance) in your **data science** pipeline, Spark also supports Sampling (https://en.wikipedia.org/wiki/Sampling_(statistics)) in general.

In the example below, we will use the **sample()** method to create a sample of the tags dataframe. The sample() method takes the following parameters:

- with replacement = true
- number of rows to sample = 20%
- a random seed = 37L

```
// Sampling With Replacement
val dfTagsSample = dfTags.sample(true, 0.2, 37L)
println(s"Number of rows in sample dfTagsSample = ${dfTagsSample.count()}")
println(s"Number of rows in dfTags = ${dfTags.count()}")
```

You should see the following output when you run your Scala application in IntelliJ:

```
Number of rows in sample dfTagsSample = 1948
Number of rows in dfTags = 9999
```

NOTE:

- If you need sampling **without** replacement, you can reuse the same sample() method but you will have to set its first parameter to **false**.

DataFrame Operations Introduction

The examples in this section will make use of the **Context** trait which we've created in Bootstrap a SparkSession (http://allaboutscala.com/big-data/spark/#bootstrap-sparksession). By extending the Context trait, we will have access to a **SparkSession**.

```
object DataFrameOperations extends App with Context {
```

Setup DataFrames

By now you should be familiar with how to create a dataframe from reading a CSV file (http://allaboutscala.com/big-data/spark/#create-dataframe-read-csv). Similar to the previous examples, we will create two dataframes, one for the StackOverflow **tags** dataset and the other for the **questions** dataset.

Note also that we will only take a subset of the questions dataset using the filter method (http://allaboutscala.com/big-data/spark/#dataframe-operations) and join method (http://allaboutscala.com/big-data/spark/#dataframe-join) so that it is easier to work with the examples in this section.

```
  val dfTags = sparkSession
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("src/main/resources/question_tags_10K.csv")
    .toDF("id", "tag")

  dfTags.show(10)

  val dfQuestionsCSV = sparkSession
    .read
    .option("header", false)
    .option("inferSchema", true)
    .option("dateFormat","yyyy-MM-dd HH:mm:ss")
    .csv("src/main/resources/questions_10K.csv")
    .toDF("id", "creation_date", "closed_date", "deletion_date", "score", "owner_userid", "answer_count")

  val dfQuestions = dfQuestionsCSV
    .filter("score > 400 and score < 410")
    .join(dfTags, "id")
    .select("owner_userid", "tag", "creation_date", "score")
    .toDF()

  dfQuestions.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+-------------------+
| id|                tag|
+---+-------------------+
|  1|               data|
|  4|                 c#|
|  4|           winforms|
|  4|    type-conversion|
|  4|            decimal|
|  4|            opacity|
|  6|               html|
|  6|                css|
|  6|               css3|
|  6|internet-explorer-7|
+---+-------------------+
only showing top 10 rows

+------------+-------------------+--------------------+-----+
|owner_userid|                tag|       creation_date|score|
+------------+-------------------+--------------------+-----+
|         131|              xdebug|2008-08-03T23:18:21Z|  405|
|         131|            phpstorm|2008-08-03T23:18:21Z|  405|
|         131|           debugging|2008-08-03T23:18:21Z|  405|
|         131|             eclipse|2008-08-03T23:18:21Z|  405|
|         131|                 php|2008-08-03T23:18:21Z|  405|
|          NA|                 osx|2008-08-05T05:39:36Z|  408|
|          NA|                 ios|2008-08-05T05:39:36Z|  408|
|          NA|         objective-c|2008-08-05T05:39:36Z|  408|
|          NA|              iphone|2008-08-05T05:39:36Z|  408|
|         122|illegalargumentex...|2008-08-06T19:26:30Z|  402|
+------------+-------------------+--------------------+-----+
only showing top 10 rows
```

Convert DataFrame row to Scala case class

With the DataFrame **dfTags** in scope from the setup section, let us show how to convert each **row** of dataframe to a Scala case class (http://allaboutscala.com/tutorials/chapter-3-beginner-tutorial-using-classes-scala/scala-tutorial-learn-define-use-case-class/).

We first create a **case class** to represent the tag properties namely **id** and **tag**.

```
case class Tag(id: Int, tag: String)
```

The code below shows how to convert each row of the dataframe dfTags into **Scala case class** Tag created above.

```
import sparkSession.implicits._
val dfTagsOfTag: Dataset[Tag] = dfTags.as[Tag]
dfTagsOfTag
  .take(10)
  .foreach(t => println(s"id = ${t.id}, tag = ${t.tag}"))
```

You should see the following output when you run your Scala application in IntelliJ:

```
id = 1, tag = data
id = 4, tag = c#
id = 4, tag = winforms
id = 4, tag = type-conversion
id = 4, tag = decimal
id = 4, tag = opacity
id = 6, tag = html
id = 6, tag = css
id = 6, tag = css3
id = 6, tag = internet-explorer-7
```

NOTE:

- We've imported **sparkSession.implicits._** which will do encoding of default types.
- To demonstrate that each row of the dataframe was mapped to a **Scala case class**, we've sliced the dataset using the **take() method** and then used the **foreach() method** which gives you access to each row of type **Tag** created above.

DataFrame row to Scala case class using map()

In the previous example, we showed how to convert DataFrame row to Scala case class (http://allaboutscala.com/big-data/spark/#dataframe-convert-row-scala-case-class) using **as[]**. If you need to manually parse each row, you can also make use of the **map() method** to convert DataFrame rows to a **Scala case class**.

In the example below, we will parse each row and normalize **owner_userid** and the **creation_date** fields. To start with, let us create a Case Class (http://allaboutscala.com/tutorials/chapter-3-beginner-tutorial-using-classes-scala/scala-tutorial-learn-define-use-case-class/) to represent the StackOverflow **question** dataset.

```
case class Question(owner_userid: Int, tag: String, creationDate: java.sql.Timestamp, score: Int)
```

Next, we'll create some functions to map each **org.apache.spark.sql.Row** into the Question case class above. Note also that we've defined Nested Functions (http://allaboutscala.com/tutorials/chapter-3-beginner-tutorial-using-functions-scala/scala-tutorial-learn-create-nested-function/) to normalize **owner_userid** and **creation_date** fields.

```
// create a function which will parse each element in the row
  def toQuestion(row: org.apache.spark.sql.Row): Question = {
    // to normalize our owner_userid data
```

```
    val IntOf: String => Option[Int] = _ match {
      case s if s == "NA" => None
      case s => Some(s.toInt)
    }

    import java.time._
    val DateOf: String => java.sql.Timestamp = _ match {
      case s => java.sql.Timestamp.valueOf(ZonedDateTime.parse(s).toLocalDateTime)
    }

    Question (
      owner_userid = IntOf(row.getString(0)).getOrElse(-1),
      tag = row.getString(1),
      creationDate = DateOf(row.getString(2)),
      score = row.getString(3).toInt
    )
  }
```

To convert each row in the DataFrame into the Question **case class**, we can then call the map() method and pass in the **toQuestion()** function which we defined above.

```
// now let's convert each row into a Question case class
  import spark.implicits._
  val dfOfQuestion: Dataset[Question] = dfQuestions.map(row => toQuestion(row))
  dfOfQuestion
    .take(10)
    .foreach(q => println(s"owner userid = ${q.owner_userid}, tag = ${q.tag}, creation date = ${q.creationDate}, score
```

You should see the following output when you run your Scala application in IntelliJ:

```
owner userid = 131, tag = xdebug, creation date = 2008-08-03 23:18:21.0, score = 405
owner userid = 131, tag = phpstorm, creation date = 2008-08-03 23:18:21.0, score = 405
owner userid = 131, tag = debugging, creation date = 2008-08-03 23:18:21.0, score = 405
owner userid = 131, tag = eclipse, creation date = 2008-08-03 23:18:21.0, score = 405
owner userid = 131, tag = php, creation date = 2008-08-03 23:18:21.0, score = 405
owner userid = -1, tag = osx, creation date = 2008-08-05 05:39:36.0, score = 408
owner userid = -1, tag = ios, creation date = 2008-08-05 05:39:36.0, score = 408
owner userid = -1, tag = objective-c, creation date = 2008-08-05 05:39:36.0, score = 408
owner userid = -1, tag = iphone, creation date = 2008-08-05 05:39:36.0, score = 408
owner userid = 122, tag = illegalargumentexception, creation date = 2008-08-06 19:26:30.0, score = 402
```

Create DataFrame from collection

So far we have seen how to create a dataframe by reading CSV file. In this example, we will show how to create a dataframe from a Collection sequence (http://allaboutscala.com/tutorials/chapter-6-beginner-tutorial-using-scala-immutable-collection/).

```
  val seqTags = Seq(
    1 -> "so_java",
    1 -> "so_jsp",
    2 -> "so_erlang",
    3 -> "so_scala",
    3 -> "so_akka"
  )

  import spark.implicits._
  val dfMoreTags = seqTags.toDF("id", "tag")
  dfMoreTags.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+---------+
```

```
| id|      tag|
+---+---------+
|  1|  so_java|
|  1|   so_jsp|
|  2|so_erlang|
|  3| so_scala|
|  3|  so_akka|
+---+---------+
```

DataFrame Union

To merge two dataframes together you can make use of the **union() method**. In this example, we will merge the dataframe **dfTags** and the dataframe **dfMoreTags** which we created from the previous section.

```
  val dfUnionOfTags = dfTags
    .union(dfMoreTags)
    .filter("id in (1,3)")
  dfUnionOfTags.show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+--------+
| id|     tag|
+---+--------+
|  1|    data|
|  1| so_java|
|  1|  so_jsp|
|  3|so_scala|
|  3| so_akka|
+---+--------+
```

DataFrame Intersection

To find the intersection between two dataframes, you can make use of the **intersection() method**. In this example, we will find the intersection between the dataframe **dfMoreTags** and the dataframe **dfUnionOfTags** which we created from the previous section.

```
  val dfIntersectionTags = dfMoreTags
    .intersect(dfUnionOfTags)
    .show(10)
```

You should see the following output when you run your Scala application in IntelliJ:

```
+---+--------+
| id|     tag|
+---+--------+
|  3|so_scala|
|  3| so_akka|
|  1| so_java|
|  1|  so_jsp|
+---+--------+
```

Append column to DataFrame using withColumn()

When running data analysis, it can be quite handy to know how to add columns to dataframe. Using the **withColumn()** method, you can easily append columns to dataframe.

In the create dataframe from collection example above, we should have dataframe **dfMoreTags** in scope. We will call the **withColumn()** method along with **org.apache.spark.sql.functions.split()** method to split the value of the tag column and create two additional columns named **so_prefix** and **so_tag**.

Note also that we are showing how to call the **drop()** method to drop the temporary column **tmp**.

```
import org.apache.spark.sql.functions._
val dfSplitColumn = dfMoreTags
  .withColumn("tmp", split($"tag", "_"))
  .select(
    $"id",
    $"tag",
    $"tmp".getItem(0).as("so_prefix"),
    $"tmp".getItem(1).as("so_tag")
).drop("tmp")
dfSplitColumn.show(10)
```

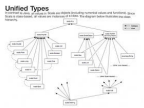
You should see the following output when you run your Scala application in IntelliJ:

```
 +---+---------+---------+------+
| id|      tag|so_prefix|so_tag|
+---+---------+---------+------+
|  1|  so_java|       so|  java|
|  1|   so_jsp|       so|   jsp|
|  2|so_erlang|       so|erlang|
|  3| so_scala|       so| scala|
|  3|  so_akka|       so|  akka|
+---+---------+---------+------+
```


Nadim Bahadoor (http://allaboutscala.com/nadim-bahadoor-founder-allaboutscala)


(http://allaboutscala.com/nadim-bahadoor-founder-allaboutscala) Senior Software Developer|Nephila Capital (https://nephila.com) Founder of allaboutscala.com (/). I have over 10 years of experience in building large scale real-time trading systems in the financial industry. Passionate about Distributed Systems, Scala, Big Data and Functional Programming. Stay in touch for upcoming tutorials!

Related Tutorials

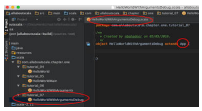


Scala Tutorial - Learn The Scala Class And Type...

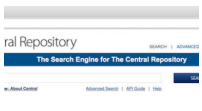
Scala Tutorial - Scala functional programming...

Scala Tutorial - Your first Scala Hello World...

Scala Tutorial - What is Scala programming...



IntelliJ Debug Configuration - Debugging Your



Ir D S

Ir D G

Other allaboutscala.com tutorials you may like: