

## Exercise 7: Broadcast variables

Here we have the maker space data across the UK. The first row is the header. This data set includes the name of the maker space email address postcard number of visitors etc. We want to answer the following question.

- How are those makers space distributed across different regions in the UK.

However, we only have the postcode for each maker space.

We don't know the region of each maker's space to answer the above question.

We need to introduce another dataset here.

We also have the UK postcode data for each postcode prefix we can find out which region it belongs to.

Combining those two data sets we should be able to answer the question how are those makers spaces distributed across different regions in the UK.

Let us create a Scala class as `UkMakerSpaces` in the package `"com.sparkTutorial.advanced.broadcast"` and paste the below code:

```
package com.sparkTutorial.advanced.broadcast

import com.sparkTutorial.commons.Utills
import org.apache.log4j.{Level, Logger}
import org.apache.spark.{SparkConf, SparkContext}

import scala.io.Source

object UkMakerSpaces {

  def main(args: Array[String]) {
    Logger.getLogger("org").setLevel(Level.ERROR)
    val conf = new
SparkConf().setAppName("UkMakerSpaces").setMaster("local[1]")
    val sparkContext = new SparkContext(conf)

    val postCodeMap = sparkContext.broadcast(loadPostCodeMap())

    val makerSpaceRdd = sparkContext.textFile("in/uk-makerspaces-
identifiable-data.csv")

    val regions = makerSpaceRdd
      .filter(line => line.split(Utills.COMMA_DELIMITER, -1)(0) !=
"Timestamp")
      .filter(line => getPostPrefix(line).isDefined)
      .map(line =>
postCodeMap.value.getOrElse(getPostPrefix(line).get, "Unknown"))

    for ((region, count) <- regions.countByValue()) println(region +
" : " + count)
```

```
}  
  
def getPostPrefix(line: String): Option[String] = {  
  val splits = line.split(Utils.COMMA_DELIMITER, -1)  
  val postcode = splits(4)  
  if (postcode.isEmpty) None else Some(postcode.split(" ")(0))  
}  
  
def loadPostCodeMap(): Map[String, String] = {  
  Source.fromFile("in/uk-postcode.csv").getLines.map(line => {  
    val splits = line.split(Utils.COMMA_DELIMITER, -1)  
    splits(0) -> splits(7)  
  }).toMap  
}  
}
```



## Exercise 8: Dataframe, Datasets and Spark SQL

Two ways to create Spark Dataframes:

Creating DataFrames from Existing RDD:

1. Infer schema by Reflection
  - a. Convert RDD containing case classes.
  - b. Use when schema is known.
2. Construct schema programmatically
  - a. Schema is dynamic
  - b. Number of fields in case class is more than 22 fields.

### TWO WAYS TO DEFINE A SCHEMA

#### 1. Define Schema Programmatically

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.functions.{col, expr}

val jsonFile = "file:///home/cloudera/Desktop/SW/in/Data/blogs.json"

// Define our schema programmatically

val schema = StructType(Array(StructField("Id", IntegerType, false),
  StructField("First", StringType, false),
  StructField("Last", StringType, false),
  StructField("Url", StringType, false),
  StructField("Published", StringType, false),
  StructField("Hits", IntegerType, false),
  StructField("Campaigns", ArrayType(StringType), false)))

//Create a DataFrame by reading from the JSON file a predefined Schema

val blogsDF = spark.read.schema(schema).json(jsonFile)

//show the DataFrame schema as output

blogsDF.show(truncate = false)

// print the schemas
```

```

print(blogsDF.printSchema)
print(blogsDF.schema)

// Show columns and expressions
blogsDF.select(expr("Hits") * 2).show(2)
blogsDF.select(col("Hits") * 2).show(2)
blogsDF.select(expr("Hits * 2")).show(2)

// show heavy hitters
blogsDF.withColumn("Big Hitters", (expr("Hits > 10000"))).show()

// Concatenate three columns, create a new column, and show the
// newly created concatenated column
blogsDF.withColumn("AuthorsId", (concat(expr("First"), expr("Last"),
expr("Id")))).select(col("AuthorsId")).show(4)


// Sort by column "Id" in descending order
blogsDF.sort(col("Id").desc).show()
blogsDF.sort($"Id".desc).show()

// In Scala to save as a Parquet file
val parquetPath = <path>
blogsDF.write.format("parquet").save(parquetPath)

```

**Note:** the expressions `blogs_df.sort(col("Id").desc)` and `blogsDF.sort($"Id".desc)` are identical. They both sort the DataFrame column named `Id` in descending order: one uses an explicit function, `col("Id")`, to return a `Column` object, while the other uses `$` before the name of the column, which is a function in Spark that converts column named `Id` to a `Column`.

**Assignment 1 :** let's read a large CSV file containing data on San Francisco Fire Department calls. we will define a schema for this file and use the `DataFrameReader` class and its methods to tell Spark what to do. Because this file contains 28 columns and over 4,380,660 records,<sup>2</sup> it's more efficient to define a schema than have Spark infer it.

Hint:

```
val sfFireFile="/path/sf-fire-calls.csv"
```

```
val fireDF = spark.read.schema(fireSchema)
  .option("header", "true")
  .csv(sfFireFile)
```

- What were all the different types of fire calls in 2018?
- What months within the year 2018 saw the highest number of fire calls?
- Which neighborhood in San Francisco generated the most fire calls in 2018?
- Which neighborhoods had the worst response times to fire calls in 2018?
- Which week in the year in 2018 had the most fire calls?
- Is there a correlation between neighborhood, zip code, and number of fire calls?
- How can we use Parquet files or SQL tables to store this data and read it back?

## 2. Infer schema by Reflection

let's look at a collection of readings from Internet of Things (IoT) devices in a JSON file (we use this file in the end-to-end example later in this section).

Let us create Case class for the file

```
case class DeviceIoTData (battery_level: Long, c02_level: Long,
cca2: String, cca3: String, cn: String, device_id: Long,
device_name: String, humidity: Long, ip: String, latitude: Double,
lcd: String, longitude: Double, scale:String, temp: Long,
timestamp: Long)

val ds =
spark.read.json("file:///home/cloudera/Desktop/SW/in/Data/iot_devices.json").as[DeviceIoTData]

val filterTempDS = ds.filter({d => {d.temp > 30 && d.humidity > 70}})
filterTempDS.show(5, false)

// In Scala
```

```

case class DeviceTempByCountry(temp: Long, device_name: String, device_id:
Long,
  cca3: String)
val dsTemp = ds
  .filter(d => {d.temp > 25})
  .map(d => (d.temp, d.device_name, d.device_id, d.cca3))
  .toDF("temp", "device_name", "device_id", "cca3")
  .as[DeviceTempByCountry]
dsTemp.show(5, false)

```

Alternatively, you could express the same query using column names and then cast to a `Dataset[DeviceTempByCountry]`:

```

// In Scala
val dsTemp2 = ds
  .select($"temp", $"device_name", $"device_id", $"device_id", $"cca3")
  .where("temp > 25")
  .as[DeviceTempByCountry]

```

**Assignment 2:** Using the Dataset API, try to execute to do the following:

- Detect failing devices with battery levels below a threshold.
- Identify offending countries with high levels of CO2 emissions.
- Compute the min and max values for temperature, battery level, CO2, and humidity.
- Sort and group by average temperature, CO2, humidity, and country.

## Using Spark SQL with queries

We will here work on the Airline On-Time Performance and Causes of Flight Delays data set, which contains data on US flights including date, delay, distance, origin, and destination. It's available as a CSV file with over a million records. Using a schema, we'll read the data into a DataFrame and register the DataFrame as a temporary view (more on temporary views shortly) so we can query it with SQL.

```
// Path to data set
val csvFile="/home/cloudera/Desktop/SW/in/Data/departuredelays.csv"

// Read and create a temporary view
// Infer schema (note that for larger files you may want to specify the schema)
val df = spark.read.format("csv").option("inferSchema", "true").option("header",
"true").load(csvFile)

// Create a temporary view
df.createOrReplaceTempView("us_delay_flights_tbl")
```

Now that we have a temporary view, we can issue SQL queries using Spark SQL. These queries are no different from those you might issue against a SQL table in, say, a MySQL or PostgreSQL database. The point here is to show that Spark SQL offers an ANSI:2003-compliant SQL interface, and to demonstrate the interoperability between SQL and DataFrames.

The US flight delays data set has five columns:

- The `date` column contains a string like `02190925`. When converted, this maps to `02-19 09:25 am`.
- The `delay` column gives the delay in minutes between the scheduled and actual departure times. Early departures show negative numbers.
- The `distance` column gives the distance in miles from the origin airport to the destination airport.
- The `origin` column contains the origin IATA airport code.
- The `destination` column contains the destination IATA airport code.

With that in mind, let's try some example queries against this data set.

- First, we'll find all flights whose distance is greater than 1,000 miles:

```
spark.sql("""SELECT distance, origin, destination FROM us_delay_flights_tbl
WHERE distance > 1000 ORDER BY distance DESC""").show(10)
```

- Next, we'll find all flights between San Francisco (SFO) and Chicago (ORD) with at least a two-hour delay:

```
spark.sql("""SELECT date, delay, origin, destination FROM us_delay_flights_tbl
WHERE delay > 120 AND ORIGIN = 'SFO' AND DESTINATION = 'ORD'
ORDER by delay DESC""").show(10)
```

- **Assignment:** It seems there were many significantly delayed flights between these two cities, on different dates. (As an exercise, convert the date column into a readable format and find the days or months when these delays were most common. Were the delays related to winter months or holidays?)
- Let's try a more complicated query where we use the CASE clause in SQL. In the following example, we want to label all US flights, regardless of origin and destination, with an indication of the delays they experienced: Very Long Delays (> 6 hours), Long Delays (2–6 hours), etc. We'll add these human-readable labels in a new column called Flight\_Delays:

```
spark.sql("""SELECT delay, origin, destination,
CASE
  WHEN delay > 360 THEN 'Very Long Delays'
  WHEN delay > 120 AND delay < 360 THEN 'Long Delays'
  WHEN delay > 60 AND delay < 120 THEN 'Short Delays'
  WHEN delay > 0 and delay < 60 THEN 'Tolerable Delays'
  WHEN delay = 0 THEN 'No Delays'
  ELSE 'Early'
END AS Flight_Delays
FROM us_delay_flights_tbl
ORDER BY origin, delay DESC""").show(10)
```



---

## Overview

In this lab, we will look at several transformations and optimizations on Dataframes and Datasets.

## Builds on



Previous labs for the transformations we'll use.

## Run time

20-30 minutes

### ➤ Explore the Spark shell UI Tasks

Open the FireFox browser (In cloudera dektop, **Applications Menu | FireFox**), and go to the launch **Spark\_UI (localhost:4040)**

In the page that opens, look for the application SEP

That should open the Spark UI Below is a sample screen shot of the UI



**NOTE:** In all screen shots, you might see differences in job ids, stage ids and other details that depend on the exact state of your Spark shell. This is because the shell used for the screen shots is likely to have executed a different overall sequence of transformations than your shell since it was started. This is not important - the important details should be easily recognizable.

## Lab: 10.1

## Spark SQL Basics

### Overview

In this lab, we will use some of the basic functionality of Spark SQL, including:

Read and write data files in varying formats Create DataFrames

### Run time

15-20 minutes

### Lab Preparation

### Tasks

Load data from the following files, if you haven't already

- *people.json* as JSON
- *wiki-pageviews.txt* as text
- *github.json* as JSON

```
//Scala Code
***
*** Load Text and JSON Data ***
***

Read people.json
> val folksDF=spark.read.json("data/people.json")

Read wiki-pageviews.txt
> var viewsDF = spark.read.text("data/wiki-pageviews.txt")

Display some of the data
> folksDF.limit(5).show
> viewsDF.limit(5).show

***
*** Read More Complex Data ***
***

Read/display data in github.json

> val githubDF=spark.read.json("data/github.json")
> githubDF.limit(5).show

***
*** Write Data ***
***
```

```
Write out folksDF as parquet file
> folksDF.write.parquet("data/people.parquet")
```

```
Write out folksDF as a CSV file
> folksDF.coalesce(1).write.option("header", true).csv("data/people.csv")
```

## View the Schema

```
[scala> githubDF.printSchema()
root
 |-- actor: struct (nullable = true)
 |   |-- avatar_url: string (nullable = true)
 |   |-- gravatar_id: string (nullable = true)
 |   |-- id: long (nullable = true)
 |   |-- login: string (nullable = true)
 |   |-- url: string (nullable = true)
 |-- created_at: string (nullable = true)
 |-- id: string (nullable = true)
 |-- org: struct (nullable = true)
 |   |-- avatar_url: string (nullable = true)
 |   |-- gravatar_id: string (nullable = true)
 |   |-- id: long (nullable = true)
 |   |-- login: string (nullable = true)
 |   |-- url: string (nullable = true)
 |-- payload: struct (nullable = true)
 |   |-- action: string (nullable = true)
 |   |-- before: string (nullable = true)
 |   |-- comment: struct (nullable = true)
 |   |   |-- _links: struct (nullable = true)
 |   |   |   |-- html: struct (nullable = true)
 |   |   |   |   |-- href: string (nullable = true)
 |   |   |   |-- pull_request: struct (nullable = true)
 |   |   |   |   |-- href: string (nullable = true)
 |   |   |   |-- self: struct (nullable = true)
 |   |   |   |   |-- href: string (nullable = true)
 |   |   |-- body: string (nullable = true)
 |   |   |-- commit_id: string (nullable = true)
 |   |   |-- created_at: string (nullable = true)
 |   |   |-- diff_hunk: string (nullable = true)
 |   |   |-- html_url: string (nullable = true)
```

The githubDF schema is quite complex. We'll look at some ways of dealing with it soon.

## Declare a Schema Explicitly Tasks

Declare a schema for folksDF. We illustrate this below. We also supply this code in a below snippet - *peopleSchema.txt* :

The file is stored in the Datasets.

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

```
val mySchema = (new StructType).add("name", StringType).add("gender",  
StringType).add("age", IntegerType)
```

Read data/people.json again, but this time supply the schema you declared.

```
> var folksDF = spark.read.schema(mySchema).json("data/people.json")
```

```
# Python
```

```
from pyspark.sql.types import *
```

```
mySchema =
```

```
StructType().add("name",StringType()).add("gender",StringType()).add("age",  
IntegerType())
```

```
folksDF = spark.read.schema(mySchema).json("data/people.json")
```

## **Lab 10.2**

### **DataFrame Transformations**

#### **Overview**

In this lab, we will perform a number of transformations on dataframes. This will provide exposure to the API, and some experience in using it.

We will cover a number of common techniques, but will not attempt to cover all

of the API, or try to illustrate every possible technique. The API is too large for that.

We'll be working at the Spark shell for all of these.

## Run time

30-40 minutes

---

### Assignment:

- Display the data. Filter on age and display the data.
- Filter on gender and display the data.
- Count how many "F" and "M" items there are.
- **[Optional]** Find the oldest person with gender "F".
  - Use SQL to write this query - e.g. `spark.sql(" ... ")`
  - Remember to register a temporary table.
  - Use a subquery to find the maximum age.

### Solution:

\*\*\*

\*\*\* Simple Transformations \*\*\*

\*\*\*

Read people.json if you haven't already

```
> val folksDF=spark.read.json("data/people.json")
```

Display the data.

```
> folksDF.show
```

Filter on age and display the data

Filter on gender and display the data

```
> folksDF.filter('age>25').show // With age greater than 25
```

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

```
| age | gender | name |
```

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

```
| 35 |    M | John |
```

```
| 40 |    F | Jane |
```

```
| 52 |    F |  Sue |
```

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

```
> folksDF.filter('age>25 && 'age<50').show // With age greater than 25 and less than 50
```

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

```
| age | gender | name |
```

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

```
| 35 |    M | John |
```

```
| 40 |    F | Jane |
```

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

```
> folksDF.filter('gender === "F").show
```

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

```
| age | gender | name |
```

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

```
| 40 |    F | Jane |
```

```
| 52 |    F |  Sue |
```

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

Count how many "F" and "M" items there are.

```
> folksDF.groupBy('gender').count.show
```

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

```
| gender | count |
```

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

```
|    F |    2 |
```

```
|    M |    2 |
```

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

[Optional] Find the oldest person with gender "F"

```
> folksDF.createOrReplaceTempView("people")
```

```
> spark.sql("SELECT name, age FROM people WHERE gender = 'F' and age =  
(select max(age) FROM people WHERE gender='F')").show
```

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

```
| name | age |
```

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

```
| Sue | 52 |
```

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

---

## Working with More Complex Data

### Tasks

- Read *github.json* if you haven't already

```
> val githubDF=spark.read.json("data/github.json")
```

- Look at the schema and 5 sample rows (`limit(5).show`) again.
- Look at the `githubDF` schema and a few rows (`limit().show`) again.
- Select the login value of the actor and display it.
- Find out how many unique logins there are in the data.
- Each row in this data, contains a type column. Display all the unique values for the 'type' column.
- Determine how many rows have a type of `CreateEvent`

```
> githubDF.limit(5).show
```

Select the actor column and view it and its schema

```
> githubDF.limit(5).select('actor').show(false)
```

```
> githubDF.select('actor').printSchema
```

Select the login value of the actor and display it

```
> actorDF.select("actor.login").limit(5).show
```

Find out how many unique logins there are in the data.

```
> actorDF.select("actor.login").dropDuplicates.count
```



Retrieve all the unique values for the 'type' column.

```
> gitHubDF.dropDuplicates("type").count
```

```
> gitHubDF.dropDuplicates("type").show
```

---

## Working with Text Data

### Tasks

- Read *wiki-pageviews.txt* if you haven't already
- Show 5 rows from this data so you can examine it.

```
val viewsDF=spark.read.text("<path>/wiki-pageviews.txt")
```

```
viewsDF.limit(5).show
```

- We can see that each row contains a single string.
  - Unlike JSON data, there is no automatic application of a schema.
  - This is a cumbersome format to work with, and we'll see ways to transform it into an easier to use format later.

### Summary

We've practiced using some of the common transformations that are available for dataframes. We'll continue using this data to delve into Spark's capabilities.



## Lab 10.3: The Dataset Typed API

### Overview

In this lab, we will work with the Dataset typed API which provides compile-time type safety.

Continue working in your spark shell in this lab as previously. You'll work with some of the dataframes created in the previous lab.

### What About Python?

Python does NOT have Dataset . It is a dynamically typed language, and thus the strongly typed Dataset makes no sense for it.

### Run time

30-40 minutes

---

### Music Data File

We've provided a small and simple JSON data file (*<path>/data/music.json*) containing data about music items. We'll use it to do our Dataset work in this lab. It has four pieces of data in it:

- title: Title of the item.
- artist: Artist who released the item.
- price: Price of the item<sup>[1]</sup><sub>SEP</sub>
- category: The music category (e.g. Pop)

### Tasks

- Create a DataFrame by reading in the music data in *data/music.json*
- View the schema of this DataFrame.
- Display the data in the DataFrame (show).

#### Simple Dataset Usage Tasks

- Define a MusicItem case class suitable for our *music.json* data<sup>[1]</sup><sub>SEP</sub> Using this class, create a typed Dataset from the musicDF dataframe created with this data in the previous lab.
- What type is musicDS? Display the data in the dataset.

### Solution:

\*\*\*

\*\*\* Music Data File \*\*\*

\*\*\*

```
val musicDF=spark.read.json("data/music.json")
```

```
> musicDF.schema
```

```
> musicDF.show
```

\*\*\*

\*\*\* Simple DataSet Usage \*\*\*

\*\*\*

Define a MusicItem case class suitable for our music.json data.

```
> case class MusicItem (title: String, artist: String, category: String, price: Double)
```

Using this class, create a typed DataSet from the musicDF dataframe.

```
> val musicDS=musicDF.as[MusicItem]
```

What type is folksDS?

DataSet of MusicItem, as shown below.

```
> musicDS
```

Display the data in the dataset.

```
> musicDS.show
```

## Compare: DataFrame vs. Dataset

We'll perform some operations on our DataFrame and Dataset representations, to get a feel of the difference.

### Tasks

- Filter on category
  - Using musicDF get all the items in the "Pop" category.
  - Using musicDS get all the items in the "Pop" category.
- Get lowest priced item in each category.
  - You'll need to group, then do an aggregation.
  - Using musicDF and untyped transformations, get the lowest price item in a category.
  - Using musicDS and typed transformations, get the lowest price item in a category.
- Transform data so that the price is reduced 10%. (You can do this by multiplying it by 0.9).
  - Using musicDF (the DataFrame) transform it to a DataFrame where the price is reduced by 10%.
    - Hint use a select and a literal to create the new price values.
  - Using musicDS (the Dataset) transform it to a Dataset where the price is doubled.
    - Use map()
    - Make sure you know how your case class is defined (i.e. what order the fields appear).
- Make a mistake - watch how errors are caught.
  - Using musicDF (the DataFrame) make a mistake in transforming it when you modify the price.
    - Try multiplying the price by a literal string - e.g. lit("A")
    - What happens?
  - Using musicDS (the Dataset) make a mistake in transforming when you modify the price.
    - Try multiplying the price by a string (e.g. "A")
    - What happens?

### Solution:

```
***  
*** Compare: DataFrame vs. DataSet ***  
***
```

```

-- Filter on category.
> musicDF.filter('category === "Pop").show

> musicDS.filter(mi => mi.category == "Pop").show

-- Using musicDF and untyped transformations, get the lowest price item
in a category.

> musicDF.groupBy('category').min("price").show

> musicDS.groupByKey(mi=>
mi.category).agg(min('price').as[Double]).show

- Transform data so that the price is reduced 10%. (You can do this by
multiplying it by 0.9).

// DataFrame
> musicDF.select('title, 'artist, 'category, 'price*lit(0.9)).show

-- Make a mistake - watch how errors are caught.

// DataFrame - error not caught - get erroneous results.

a> musicDF.select('title, 'artist, 'category, 'price*lit("A")).show

// DataSet - error caught at parsing time.
> musicDS.map (mi => MusicItem(mi.title, mi.artist, mi.category,
mi.price*"A")).show
> musicDS.map (mi => MusicItem(mi.title, mi.artist, mi.category,
mi.price*"A")).show

```

## Summary

We've practiced some fairly simple transformations with Datasets. We've also done the same thing with DataFrames to get a feel of the difference.

DataFrames are a little easier to program to. However, as we saw, Datasets can catch errors earlier than DataFrames. This can be important when debugging and maintaining a large system.



## Lab 10.4: Splitting Text Data

### Overview

In this lab, we'll work with text data. The data is regularly structured, but since it's in text format, Spark can't deduce the structure on its own.

We'll use some of the DataFrame tools to create a DataFrame with a schema that's easy to use for querying the data.

The data we'll use contains page view data from Wikimedia.

### Run time

30-40 minutes

---

### Wikimedia PageView Data File

We provide a data file (*<path>/data/wiki-pageviews.txt*) that contains a dump of pageview data for many projects under the Wikimedia umbrella.

The data file has lines containing four fields.

1. Domain.project (e.g. "en.b")
2. Page name (e.g. "AP\_Biology/Evolution")
3. Page view count (e.g. 1)
4. Total response size in bytes (e.g. 10662 - but for this particular dump, value is always 0).

The data is in simple text format, so when we read it in, we get a dataframe with a single column - a string containing all the data in each row. This is cumbersome to work with. In this lab, we'll apply a better schema to this data.

- **Tasks** Create a DataFrame by reading in the page view data in `spark-labs/data/wiki-pageviews.txt`.

- Once you've created it, view a few lines to see the format of the data.
  - You'll see that you have one line of input per dataframe row.

### Split the Lines

Our first step in creating an easier to use schema is splitting each row into separate columns. We'll use the `split()` function defined in the Spark SQL functions. We've used this in our word count examples in the main manual.

### Tasks

- Create a dataframe by splitting each line up.<sup>[1]</sup>
  - Use `split()` to split on a whitespace (pattern of `"\\s+"`)

**Python:** You'll need to import from the functions module as shown in the Python example below

```
//Python
> from pyspark.sql.functions import *
> splitViewsDF = // ...
```

### Create a Better Schema

We'll create a dataframe with an easier-to-use schema containing the following columns which align with the data in the views file.

- `domain: String` - The domain.project data.
- `pageName: String` - The page name.
- `viewCount: Integer` - The view count.
- `size: Long` - The response size (always 0 in this file, but we'll keep it in our dataframe).

Execute the below queries one after the other

```
***
*** Wikimedia PageView Data File ***
***

Read the data file, and show some lines
> val viewsDF=spark.read.text("data/wiki-pageviews.txt")
> viewsDF.limit(5).show
```

\*\*\*

\*\*\* Split the Lines \*\*\*

\*\*\*

**Split on whitespace, show some lines, show the schema**

```
> val splitViewsDF = viewsDF.select(split('value, "\\s+").as("splitLine"))  
> splitViewsDF.limit(5).show(false)
```

```
> splitViewsDF.printSchema
```

\*\*\*

\*\*\* Create a Better Schema \*\*\*

\*\*\*

**Create a dataframe with a better schema, view the schema, and view some data.**

```
>val                                viewsWithSchemaDF                                =  
splitViewsDF.select('splitLine(0).as("domain"),  
'splitLine(1).as("pageName"), 'splitLine(2).cast("integer").as("viewCount"),  
'splitLine(3).cast("long").as("size"))
```

```
> viewsWithSchemaDF.printSchema
```

```
> viewsWithSchemaDF.limit(5).show
```

\*\*\*

\*\*\* Try Some Queries \*\*\*

\*\*\*

**Find rows where the viewCount > 500, and display 5 of them**

```
> viewsWithSchemaDF.filter('viewCount>500').limit(5).show
```

**Same query as above with domain of "en" - note the triple = for equality**

```
>viewsWithSchemaDF.filter('domain=="en").filter('viewCount>500').limit  
(5).show
```



**Find the 5 rows with the largest viewCount and a domain of "en"**

```
>viewsWithSchemaDF.filter('domain=== "en").orderBy('viewcount.desc').limit(5).show
```

**Find the 5 rows with the largest viewCount and a domain of "en", and where the pageName doesn't contain a colon (":")**

```
>viewsWithSchemaDF.filter('domain=== "en").filter(!'pageName.contains(":')).orderBy('viewcount.desc').limit(5).show
```

### Summary

We can see that text data can require a little more work than data like JSON with a pre-existing structure. Once you've restructured it with a clear schema, which is not usually difficult, then the full power of DataFrames can be easily applied



## Exercise 9: Dataframe, Datasets and Spark SQL Exploring Grouping

### Overview

In this lab, we will work with the Github archive that contains activity for a single day. We'll analyze it to find the top contributors for that day.

### Builds on

None - but you must have the spark shell running.

### Run time

25-35 minutes

---

### Github Activity Archive

You've already worked with the data file (*spark-labs/data/github.json*) that contains a log of all github activity for one day. In previous labs you should already have

Loaded the data, and viewed the schema.

Selected the actor column, which has a nested structure, and worked with some of its subelements.

We illustrate doing that below.

```
// Load the file into Spark
> val githubDF=spark.read.json("spark-labs/data/github.json")
// Select the actor column
> val actorDF = githubDF.select('actor')
// Print actor schema
> actorDF.printSchema
// Select the actor's login value - note how we
// Use a SQL expression in the select, not a Column
> actorDF.select("actor.login").limit(5).show
```

## Assignment: Query the Data by Actor's Login Value

### Tasks

- Query the github data for how many entries exist in the data for each actor's login. Use the DSL.

### Hints:

- You'll want to group the data by the actor's login.
- You'll probably want to use an SQL expression to express the actor's login, not a column value.
- You want a count of entries for each login value. Show a few rows of this data.
- Lastly, find the 20 logins with the largest number of contributions, and display them.

## Assignment 2: Use SQL Tasks

- Optionally, try doing the above query in SQL.
- It's pretty much standard SQL, so if you know that well, it's not very complex.
- Remember to create a temporary view (createOrReplaceTempView)

### Summary

The task we did in this lab is not overly complex, but it's also not trivial. Spark SQL makes it reasonable for us to accomplish this in a short lab, either using the DSL, or using SQL.

If you wanted to do this using RDDs, it would be a much more complex series of transformations - starting with the messy ones of parsing the JSON data. This is why Spark SQL is becoming the standard API for working with Spark.



## Exercise 10: Seeing Catalyst at Work

### Overview

In this lab, we will look at several transformations and examine the optimizations that Catalyst performs.

### Builds on

Previous labs for the transformations we'll use.

### Run time

20-30 minutes

---

### Lab Preparation

We'll first work with the Wikimedia view data, and see how Catalyst helps to optimize queries involving filtering.

### Tasks

You've already worked with the data file (*spark-labs/data/wiki-pageviews.txt*). In previous labs you should already have done the below. If you haven't, then do it now.

- Loaded the data, and then split on whitespace.
- Create a new dataframe with a finer-grained schema.
- We illustrate doing that below.

```
// Load the file into Spark
> val viewsDF=spark.read.text("spark-labs/data/wiki-pageviews.txt")

// Split on whitespace
> val splitViewsDF = viewsDF.select(split('value, "\\s+").as("splitLine"))

// Use a better schema
> val viewsWithSchemaDF = splitViewsDF.select('splitLine(0).as("domain"),
'splitLine(1).as("pageName"),          'splitLine(2).cast("integer").as("viewCount"),
'splitLine(3).cast("long").as("size"))
```

# Python

```
> from pyspark.sql.functions import *
```

```
# Load the file into Spark

> viewsDF=spark.read.text("spark-labs/data/wiki-pageviews.txt")

# Split on whitespace

>      splitViewsDF      =      viewsDF.select(split(viewsDF.value,
"\s+").alias("splitLine"))

# Use a better schema

>                                viewsWithSchemaDF                                =
splitViewsDF.select(splitViewsDF.splitLine[0].alias("domain"),splitViews
DF.splitLine[1].alias("pageName"),splitViewsDF.splitLine[2].cast("integer
").alias("viewCount"), splitViewsDF.splitLine[3].cast("long").alias("size"))
```

## Push Down Predicate

### Tasks

- First, write a transformation to order viewsWithSchemaDF by viewCount .
- explain the above transformation.
  - Note that there is shuffling involved (an Exchange ).
  - It needs to shuffle data to sort the rows.
- Next, filter viewsWithSchemaDF so you only view rows where the domain starts with "en".
  - Put the filter at the end of the transformation chain in your code (after the ordering).
  - You can use the following on the domain column to accomplish this.
    - **Scala:** startsWith("en") (Uppercase W)<sup>[1]</sup><sub>[SEP]</sub>
    - **Python:** startswith("en") (Lowercase w)
    - explain this transformation.<sup>[1]</sup><sub>[SEP]</sub>
    - You should see the filtering happening before the shuffle for ordering.
    - Catalyst has **pushed the filter down** to improve efficiency.<sup>[1]</sup><sub>[SEP]</sub>
    - View the steps that Catalyst took with the detailed version of explain()
      - **Scala:** explain(true)
      - **Python:** explain(True)<sup>[1]</sup><sub>[SEP]</sub>
- Optionally, try the transformation with the filter placed before the ordering.
  - It should give you exactly the same plan.

## Scala Only: Work with Datasets and lambdas

We'll now create a Dataset and filter it using a lambda. We'll look at how the lambda affects the Catalyst optimizations.

### Tasks

- Declare a case class for the Wikimedia data.
- Create a Dataset using the case class.

We show this code below

```
> case class WikiViews(domain:String, pageName:String, viewCount:Integer, size:Long)
> val viewsDS = viewsWithSchemaDF.as[WikiViews]
```

- Next, filter viewsDS so you only view rows where the domain starts with "en".
  - Put the filter at the end of the transformation chain (after the ordering)
  - You have to use a lambda for this.
  - You can use startsWith("en") on the domain value to accomplish this (this is a Scala function on strings).
  - explain this transformation.<sup>[1]</sup><sub>SEP</sub>

```
viewsDS.orderBy('viewCount).filter(view=>view.domain.startsWith("en")).explain
```

- Where does the filtering happen? It should be the very last thing. Why?

### Summary Catalyst rules!

We've explained it, now you see it.<sup>[1]</sup><sub>SEP</sub> But lambdas can overthrow the ruler - so be cautious with them.



---

## Exercise 11: Joins and Broadcast

### Overview

In this lab, we will explore some slightly more complex queries, including possible alternatives that use a join.

We'll also look at Catalyst's broadcast optimizations when doing joins, and examine performance with those optimizations in place and without them.

### Builds on

None

### Run time

30-40 minutes

---

### Top 20 Contributors - Straightforward Query

We want to find out the 20 contributors who have the most entries in the github data. Let's first do this in a straightforward way, as done in an earlier lab.

### Tasks

- Find the top 20 contributors as follows.

```
// Scala

> val scanQuery =
githubDF.groupBy("actor.login").count.orderBy('count.desc).limit(20)

> scanQuery.show
```

#Python

```
> scanQuery =
githubDF.groupBy("actor.login").count().orderBy("count",ascending=False).limit(
20)

> scanQuery.show()
```

- Open the Web UI (localhost:4040) and look at the jobs tab.<sup>[1]</sup><sub>[SEP]</sub>
- Note the execution time (0.6s on our system).<sup>[1]</sup><sub>[SEP]</sub>
- Drill down on this job, and note the shuffle data (333KB on our system)

---

## Top 20 Contributors - Join Query

We've decided that we want to track 20 specific contributors (the top 20 contributors from the previous month), instead of the top 20 contributors at any given moment. We keep the login values of these contributors-of-interest in another data file we supply (*spark-labs/data/github-top20.json*)

Write a join query that gives the actor login and count from the entries in *github.json* for the login values in *github-top20.json*.

### Tasks(Assignment)

- Review the data in *github-top20.json*.
- Accomplish our query needs by:
  - Loading *github-top20.json* into a dataframe (githubTop20DF)
  - Joining githubDF and githubTop20DF<sup>[1]</sup><sub>[SEP]</sub> in topContributorsJoinedDF
  - Grouping by actor.login
  - Counting the results, and ordering by descending count.

---

## Join Query Performance

Let's review the performance of this join query, and the optimizations that are being done by Catalyst.

### Tasks

- Open a browser window on the Web UI - localhost:4040.
  - Note the time taken for the last show performed above, drill through on the job for it, and note the shuffle data.
  - On our system it took 0.6s and shuffled 7.9K of data.
  - This is much less data shuffling than the scanQuery done earlier (40 times reduction).
- Execute topContributorsJoinedDF.explain to see what is going on.
  - We illustrate this below. Note the broadcasts which we have highlighted.
  - Catalyst has optimized the join to use broadcast.



```

|   jerrievesque| 29|
|   LukasReschke| 28|
|   nwt-patrick| 27|
|   Somasis| 27|
|   mikegazdag| 26|
|   tterragl098| 23|
|   EmanueleMinotto| 22|
+-----+-----+

```

```

scala> topContributorsJoinedDF.explain()
== Physical Plan ==
*(4) Sort [count#540L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#540L DESC NULLS LAST, 200)
   +- *(3) HashAggregate(keys=[actor#6.login#556], functions=[count(1)])
      +- Exchange hashpartitioning(actor#6.login#556, 200)
         +- *(2) HashAggregate(keys=[actor#6.login AS actor#6.login#556], functions=[partial_count(1)])
            +- *(2) Project [actor#6]
               +- *(2) BroadcastHashJoin [actor#6.login], [login#421], Inner, BuildRight
                  :- *(2) Project [actor#6]
                     : +- *(2) Filter isNotNull(actor#6)
                     :   +- *(2) FileScan json [actor#6] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/nehah/Desktop/Neha/Spark/spark-labs/data/github.json], PartitionFilters: [], PushedFilters: [IsNotNull(actor)], ReadSchema: struct<actor:struct<avatar_url:string,gravatar_id:string,id:bigint,login:string,url:string>>
                        +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
                           +- *(1) Project [login#421]
                              +- *(1) Filter isNotNull(login#421)
                                 +- *(1) FileScan json [login#421] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/nehah/Desktop/Neha/Spark/spark-labs/data/github-top20.json], PartitionFilters: [], PushedFilters: [IsNotNull(login)], ReadSchema: struct<login:string>

scala>

```

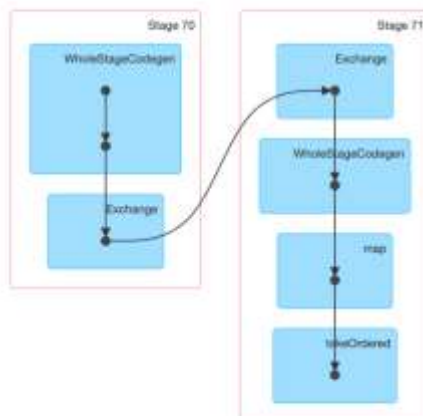
## Details for Job 36

Status: SUCCEEDED

Completed Stages: 2

Event Timeline

DAG Visualization




### Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
71	<a href="#">show id &lt;console&gt;:32</a>	<a href="#">+details</a> 2020/05/03 00:41:46	87 ms	<div><div>200/200</div></div>			9.6 KB	
70	<a href="#">show id &lt;console&gt;:32</a>	<a href="#">+details</a> 2020/05/03 00:41:46	67 ms	<div><div>11/11</div></div>	43.6 MB			9.6 KB

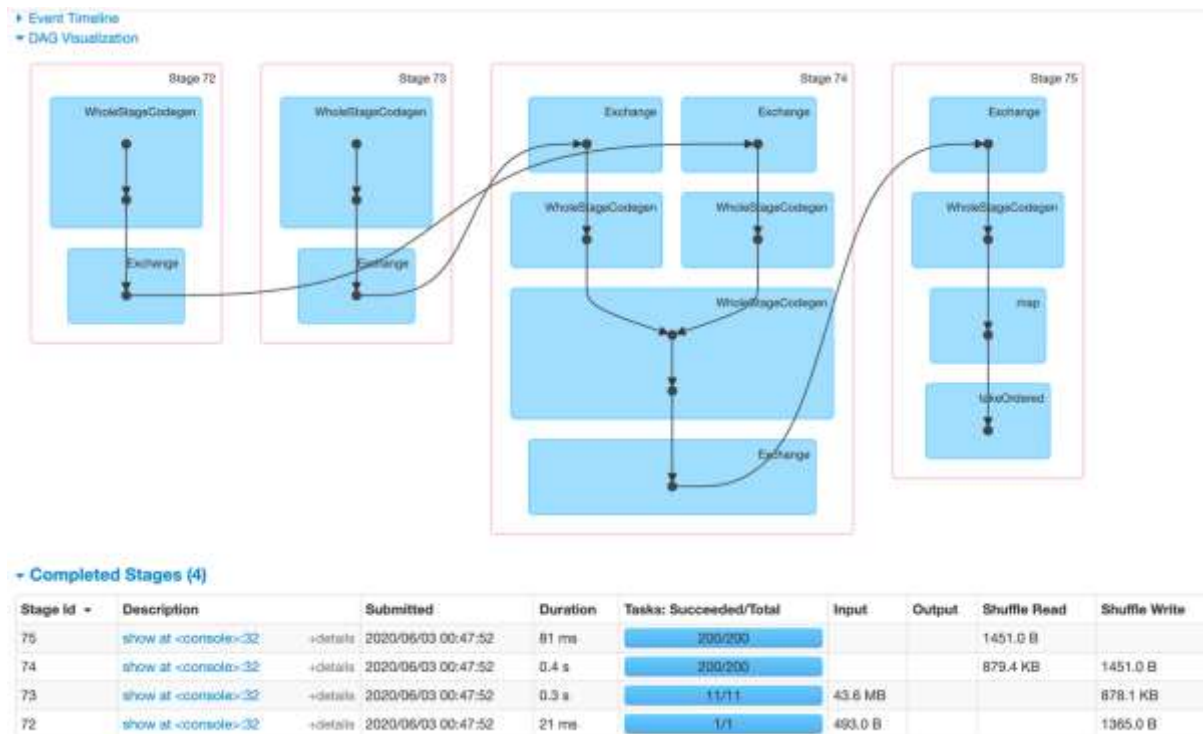
## Join Performance without Broadcast

Let's disable the Catalyst broadcast optimization and look at what happens.

## Tasks:

- Disable automatic broadcasting, by setting the appropriate configuration property as shown below.
- Next, create and show the topContributorsJoinedDF query again
  -  You must create the DataFrame after setting the config property, to make sure it conforms to the new setting.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold",-1)
```



```
scala> topContributorsJoinedDF.explain()
== Physical Plan ==
*(7) Sort [count#540L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#540L DESC NULLS LAST, 200)
   +- *(6) HashAggregate(keys=[actor#6.login#568], functions=[count(1)])
      +- Exchange hashpartitioning(actor#6.login#568, 200)
         +- *(5) HashAggregate(keys=[actor#6.login AS actor#6.login#568], functions=[partial_count(1)])
            +- *(5) Project [actor#6]
               +- *(5) SortMergeJoin [actor#6.login], [login#421], Inner
                  :- *(2) Sort [actor#6.login ASC NULLS FIRST], false, 0
                     : +- Exchange hashpartitioning(actor#6.login, 200)
                        : +- *(1) Project [actor#6]
                           : +- *(1) Filter isnotnull(actor#6)
                              : +- *(1) FileScan json [actor#6] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/nehah/Desktop/Neha/Spark/spark-labs/data/github.json], PartitionFilters: [], PushedFilters: [IsNotNull(actor)], ReadSchema: struct<actor:string,avatar_url:string,gravatar_id:string,id:bigint,login:string,url:string>
                                 +- *(4) Sort [login#421 ASC NULLS FIRST], false, 0
                                    +- Exchange hashpartitioning(login#421, 200)
                                       +- *(3) Project [login#421]
                                          +- *(3) Filter isnotnull(login#421)
                                             +- *(3) FileScan json [login#421] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/nehah/Desktop/Neha/Spark/spark-labs/data/github-top20.json], PartitionFilters: [], PushedFilters: [IsNotNull(login)], ReadSchema: struct<login:string>
scala>
```

- Observe the execution time and shuffled data. This is almost **100 times** the amount of shuffle data!
- Renewable automatic broadcasting, by setting the appropriate configuration property as shown below.

```
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold",1024*1024*10)
```

## Summary

Joins are powerful tools, but distributed joins can be expensive in terms of amount of data shuffled. We've worked with them here, and seen how Catalyst can optimize by broadcasting a small data set.

Note that we've used a common strategy here - we've replaced a query that was resource intensive (our straight scan query), with a different query (querying on a specific group of logins that are provided). The new query is not exactly equivalent, but fills our needs, and is much less resource intensive.



## Exercise 12: Hive with Spark integration



- Let's create table "reports" in the hive. I am using boa schema in which I am creating a table.

Enter in to hive CLI and use below commands to create a table:

```
CREATE schema boa;  
CREATE TABLE boa.reports(id INT,days INT,YEAR INT);  
INSERT INTO TABLE boa.reports VALUES  
(121,232,2015),(122,245,2015),(123,134,2014),(126,67,2016),(182,122,2016),(137,9  
2,2015),(101,311,2015);  
select * from boa.reports;
```

- Go to spark-shell:

Now, create a data frame hiveReports using below command:

```
var hiveReports = spark.sqlContext.sql("select * from boa.reports")
```

- Check whether dataset report is loaded into data frame hiveReport or not using below command:  
Check schema of Data Frame.
- hiveReports.show() will show the same results as "select \* from boa.reports" in Hive CLI.

➤ Accessing Catalog from spark-shell

Catalog is available on spark session. The following code shows how to access catalog.

```
val catalog = sparkSession.catalog
```

➤ Querying the databases

Once we have access to catalog, we can use it to query the databases. All the API's on catalog returns a dataset.

```
catalog.listDatabases().select("name").show()
```

➤ Registering Dataframe with createTempView

In earlier versions of spark, we used to register a dataframe using registerTempTable. But in spark 2.0, this API is deprecated. The registerTempTable API was one of the source of confusion as users used think it materializes the dataframe and saves as a temporary table which was not the case. So this API is replaced with createTempView.

createTempView can be used as follows.

```
df.createTempView("sales")  
spark.catalog.listTables("boa").show()
```

Once we have registered a view, we can query it using listTables.

➤ Checking is table cached or not

Catalog not only is used for querying. It can be used to check state of individual tables. Given a table, we can check is it cache or not. It's useful in scenarios to make sure we cache the tables which are accessed frequently.

```
catalog.isCached("sales")
```

You will get false as by default no table will be cache. Now we cache the table and query again.

```
df.cache()  
catalog.isCached("sales")
```

Now it will print true.

➤ Drop view

We can use catalog to drop views. In spark sql case, it will deregister the view. In case of hive, it will drop from the metadata store.

```
catalog.dropTempView("sales")
```

Query registered functions

Catalog API not only allow us to interact with tables, it also allows us to interact with udf's. The below code shows how to query all functions registered on spark session. They also include all built in functions.

```
catalog.listFunctions().select("name","description","className","isTemporary").show(100)
```



## Exercise 13: SQL Tables and Views

### CREATING A MANAGED TABLE

Tables reside within a database. By default, Spark creates tables under the default database. To create your own database name, you can issue a SQL command from your Spark application or notebook. Using the US flight delays data set, let's create both a managed and an unmanaged table. To begin, we'll create a database called training\_spark\_db and tell Spark we want to use that database:

// In Scala/Python

Launch

```
bin/spark-sql -S
```

for spark-sql console

```
CREATE DATABASE learn_spark_db
```

```

spark-sql> CREATE DATABASE learn_spark_db;
spark-sql> show databases;
bdp
default
demohivespark
learn_spark_db
spark-sql> █

```

Launch bin/spark-shell

```
spark.catalog.listDatabases.show(false)
```

```

scala> spark.catalog.listDatabases.show(false)
26/07/05 11:08:19 WARN DomainSocketFactory: The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.
+-----+-----+-----+
|name|description|locationUri|
+-----+-----+-----+
|bdp| |hdfs://quickstart.cloudera:8020/user/hive/warehouse/bdp.db|
|default|Default Hive database|hdfs://quickstart.cloudera:8020/user/hive/warehouse|
|demohivespark| |hdfs://quickstart.cloudera:8020/user/hive/warehouse/demohivespark.db|
|learn_spark_db| |hdfs://quickstart.cloudera:8020/user/hive/warehouse/learn_spark_db.db|
+-----+-----+-----+

```

```
spark.sql("USE training_spark_db")
```

```
spark.sql("CREATE TABLE managed_us_delay_flights_tbl (date STRING, delay INT, distance INT, origin STRING, destination STRING)")
```

```
val csvFile="file:///home/cloudera/Desktop/SW/in/Data/departuredelays.csv"
```

```
val schema= date STRING, delay INT, distance INT, origin STRING, destination STRING"
```

```
val df = spark.read.format("csv").option("schema",schema).option("header", "true").load(csvFile)
```

```
import org.apache.spark.sql.{Row, SaveMode }
```

```
df.write.mode(SaveMode.Overwrite).saveAsTable("managed_us_delay_flights_tbl")
```



```
scala> df.write.mode(SaveMode.Overwrite).saveAsTable("managed_us_delay_flights_tbl")

scala> spark.sql("select * from managed_us_delay_flights_tbl limit 5").show()
+-----+-----+-----+-----+-----+
|   date|delay|distance|origin|destination|
+-----+-----+-----+-----+-----+
|01011245|    6|    602|   ABE|        ATL|
|01020600|   -8|    369|   ABE|        DTW|
|01021245|   -2|    602|   ABE|        ATL|
|01020605|   -4|    602|   ABE|        ATL|
|01031245|   -4|    602|   ABE|        ATL|
+-----+-----+-----+-----+-----+
```

## CREATING AN UNMANAGED TABLE

By contrast, you can create unmanaged tables from your own data sources—say, Parquet, CSV, or JSON files stored in a file store accessible to your Spark application.

To create an unmanaged table from a data source such as a CSV file, in SQL use:

```
spark.sql("""CREATE TABLE us_delay_flights_tbl(date STRING, delay INT,
distance INT, origin STRING, destination STRING)
USING csv OPTIONS (PATH
'file:///home/cloudera/Desktop/SW/in/Data/departuredelays.csv')""")
```

```
df.write.mode(SaveMode.Overwrite).option("path",
"/tmp/data/us_flights_delay").saveAsTable("us_delay_flights_tbl")
```

```
scala> df.write.mode(SaveMode.Overwrite).option("path", "/tmp/data/us_flights_delay").saveAsTable("us_delay_flights_tbl")

scala> spark.sql("select * from us_delay_flights_tbl limit 5").show()
+-----+-----+-----+-----+-----+
|   date|delay|distance|origin|destination|
+-----+-----+-----+-----+-----+
|01011245|    6|    602|   ABE|        ATL|
|01020600|   -8|    369|   ABE|        DTW|
|01021245|   -2|    602|   ABE|        ATL|
|01020605|   -4|    602|   ABE|        ATL|
|01031245|   -4|    602|   ABE|        ATL|
+-----+-----+-----+-----+-----+
```

## Exercise 14: Hive with Spark integration using IntelliJ

Update pom.xml with the below dependency for hive libraries

```
<dependency>

    <groupId>org.apache.spark</groupId>
```



```
<artifactId>spark-hive_${scala.tools.version}</artifactId>

<version>${spark.version}</version>

</dependency>
```

Let us create a Scala Object in the existing workspace of training-spark, with the below provided code.

```
import java.io.File

import org.apache.spark.sql.{Row, SaveMode, SparkSession}

case class Record(key: Int, value: String)

// warehouseLocation points to the default location for managed databases and
// tables
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example") .master("local[*]")
  // .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING
hive")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO
TABLE src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()

// +---+-----+
```

```

// |key| value|
// +---+-----+
// |238|val_238|
// | 86| val_86|
// |311|val_311|
// ...

// Aggregation queries are also supported.
sql("SELECT COUNT(*) FROM src").show()
// +-----+
// |count(1)|
// +-----+
// |   500 |
// +-----+

// The results of SQL queries are themselves DataFrames and support all normal
// functions.
val sqlDF = sql("SELECT key, value FROM src WHERE key < 10 ORDER BY
key")

// The items in DataFrames are of type Row, which allows you to access each
// column by ordinal.
val stringsDS = sqlDF.map {
  case Row(key: Int, value: String) => s"Key: $key, Value: $value"
}
stringsDS.show()
// +-----+
// |          value|
// +-----+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|

```

```

// ...

// You can also use DataFrames to create temporary views within a
SparkSession.
val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val_$i")))
recordsDF.createOrReplaceTempView("records")

// Queries can then join DataFrame data with data stored in Hive.
sql("SELECT * FROM records r JOIN src s ON r.key = s.key").show()
// +---+-----+---+-----+
// |key| value|key| value|
// +---+-----+---+-----+
// | 2| val_2| 2| val_2|
// | 4| val_4| 4| val_4|
// | 5| val_5| 5| val_5|
// ...

// Create a Hive managed Parquet table, with HQL syntax instead of the Spark
SQL native syntax
// `USING hive`
sql("CREATE TABLE hive_records(key int, value string) STORED AS
PARQUET")

// Save DataFrame to the Hive managed table
val df = spark.table("src")
df.write.mode(SaveMode.Overwrite).saveAsTable("hive_records")

// After insertion, the Hive managed table has data now
sql("SELECT * FROM hive_records").show()
// +---+-----+
// |key| value|
// +---+-----+
// |238| val_238|
// | 86| val_86|

```

```

// |311|val_311|
// ...

// Prepare a Parquet data directory
val dataDir = "/tmp/parquet_data"
spark.range(10).write.parquet(dataDir)
// Create a Hive external Parquet table
sql(s"CREATE EXTERNAL TABLE hive_bigints(id bigint) STORED AS
PARQUET LOCATION '$dataDir'")
// The Hive external table should already have data
sql("SELECT * FROM hive_bigints").show()
// +---+
// | id |
// +---+
// |  0 |
// |  1 |
// |  2 |
// ... Order may vary, as spark processes the partitions in parallel.

// Turn on flag for Hive Dynamic Partitioning
spark.sqlContext.setConf("hive.exec.dynamic.partition", "true")
spark.sqlContext.setConf("hive.exec.dynamic.partition.mode", "nonstrict")
// Create a Hive partitioned table using DataFrame API
df.write.partitionBy("key").format("hive").saveAsTable("hive_part_tbl")
// Partitioned column `key` will be moved to the end of the schema.
sql("SELECT * FROM hive_part_tbl").show()
// +-----+---+
// | value|key|
// +-----+---+
// |val_238|238|
// | val_86| 86|

```

```
// |val_311|311|  
spark.stop()
```

Clean and build the jar using maven tools and submit the spark-submit command

```
bin/spark-submit --class  
com.sparkTutorial.hiveintegration.HiveIntegrationSpark  
/home/cloudera/Desktop/SW/Code/Lesson4/Lesson4/target/trainingSpark-1.0.jar -  
-master yarn
```

Observe the output from the console and hive table created from CLI.

The metastore in Hive will store, as below

```
(base) [cloudera@quickstart spark]$ hdfs dfs -ls /user/hive/warehouse  
Found 7 items  
drwxrwxrwx - cloudera supergroup 0 2020-06-03 06:03 /user/hive/warehouse/bdp.db  
drwxrwxrwx - cloudera supergroup 0 2020-06-04 01:03 /user/hive/warehouse/demohivespark.db  
drwxrwxrwx - cloudera supergroup 0 2020-06-05 23:51 /user/hive/warehouse/hive_part_tbl  
drwxr-xr-x - cloudera supergroup 0 2020-06-05 23:50 /user/hive/warehouse/hive_records  
drwxrwxrwx - cloudera supergroup 0 2020-06-05 04:55 /user/hive/warehouse/partitioned_user  
drwxrwxrwx - cloudera supergroup 0 2020-06-05 23:49 /user/hive/warehouse/src  
drwxrwxrwx - cloudera supergroup 0 2020-06-05 04:48 /user/hive/warehouse/users_part  
(base) [cloudera@quickstart spark]$ █
```



## Exercise 15: Produce and Consume Apache Kafka Messages

### Start the Zookeeper server

1. Open a new terminal window and browse to the location “/usr/lib/zookeeper” using cd. Start the zookeeper by command  
Navigate to /home/cloudera/Desktop/SW/zookeeper-3.4.14.

```
bin/zkServer.sh start
```

```
[cloudera@quickstart ~]$ cd /usr/lib/zookeeper
[cloudera@quickstart zookeeper]$ bin/zkServer.sh start
JMX enabled by default
Using config: /usr/lib/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[cloudera@quickstart zookeeper]$ █
```

2. You can now validate that Zookeeper is running correctly in standalone mode by connecting to the client port and sending the four letter command “srvr”.

```
telnet localhost 2181
```

```
[cloudera@quickstart zookeeper]$ telnet localhost 2181
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.5-cdh5.12.0--1, built on 06/29/2017 11:30 GMT
Latency min/avg/max: 0/0/1303
Received: 30491
Sent: 30604
Connections: 4
Outstanding: 0
Zxid: 0xe14
Mode: standalone
Node count: 737
Connection closed by foreign host.
[cloudera@quickstart zookeeper]$ █
```

## Starting Kafka and Creating a Kafka Topic

1. Open a new terminal and start Kafka broker, Change the user to cloudera with su - - and provide password as cloudera.

Navigate to /home/cloudera/Desktop/SW/kafka\_2.11-2.2.1

```
bin/kafka-server-start.sh config/server.properties
```

Kafka broker is running on port 9042.

- Since your exercise environment is a single-node cluster running on a virtual machine, use a replication factor of 1 and a single partition.

```
$ bin/kafka-topics.sh --create \ --zookeeper localhost:2181 \ --
replication-factor 1 \ --partitions 1 \
--topic weblogs
```

The name of the topic that you wish to create.

The number of replicas of the topic to maintain within the cluster.

The number of partitions to create for the topic.

KYVOR EDUBREEZE