Spark
Dataframes in Python

---

1 Dataset

This lab guide uses the dataset `ks-project-201801.csv`. It can be downloaded, either from the course web page or from the following link:
https://www.kaggle.com/kemical/kickstarter-projects
The previous link has also information about the dataset and the meaning of the columns in the CSV file.

2 Required software

This guide requires the following software:

- Python3
- Spark
- Java 11 SDK

3 Using Dataframes

3.1 Example: Reading and processing a CSV

Create a python program, `MyPySpark1.py`, to read the file with the dataset with the following contents:

```
from pyspark.sql import SparkSession

spark = SparkSession \
      .builder \
      .appName("PySpark example") \
      .getOrCreate()

df = spark \
      .read \
      .option("header", "true") \
      .csv("ks-projects-201801.csv")

df.printSchema()
print("== Row count: ", df.count(), " ==")
df.show()
spark.stop()
```

Run the program with
`spark-submit MyPySpark1.py`

Check the results by locating in the information output by spark:

- The dataset schema
- The number of rows found (does this value include the header row?)
- The contents of the first 20 rows

Check in the schema information that every column was a string, that can contain null values.

Let's ask spark to try to infer the actual datatypes by adding another option to the pipeline that creates a DataFrame from the dataset in the CSV file:

```
df = spark \
      .read \
      .option("header", "true") \
      .option("inferSchema", "true") \
      .csv("ks-projects-201801.csv")

      val data = spark.read
            .option("header", true)
            .option("inferSchema", true)
            .csv(filePath)
```

Re-build the package and re-run. Did it improve the datatypes found?

Note: *Another interesting option for importing tsv (Tab Separated Values) is:* `.option("sep", "\t")`

Now run the following commands:

```
$ cp MyPySpark1.py MyPySpark2.py
```

Edit `MyPySpark2.py` and improve the previous example by creating a second DataFrame where the data types of numeric and date columns are changed from String to the expected type:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import DateType

spark = SparkSession \
      .builder \
      .appName("PySpark example") \
      .getOrCreate()

df = spark \
      .read \
      .option("header", "true") \
      .csv("ks-projects-201801.csv")

dftypes = df \
      .withColumn("deadline", df["deadline"].cast(DateType())) \
      .withColumn("goal", df["goal"].cast(DoubleType())) \
      .withColumn("pledged", df["pledged "].cast(DoubleType ()))

dftypes.printSchema()
dftypes.show()

spark.stop()
```

Now repeat the same procedure and correct the data types for the remaining columns (with numbers or dates).

We can use the function filter (available from the RDDs API) to select only the rows where the deadline was between two specific dates. To do this create a new DataFrame:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import DateType
from pyspark.sql.functions import desc, sum

spark = SparkSession \
      .builder \
      .appName("PySpark example") \
```

```
        .getOrCreate()

df = spark \
        .read \
        .option("header", "true") \
        .csv("ks-projects-201801.csv")

dftypes = df \
        .withColumn("deadline", df["deadline"].cast(DateType())) \
        .withColumn("goal", df["goal"].cast(DoubleType())) \
        .withColumn("pledged", df["pledged "].cast(DoubleType ()))

dffilter = dftypes \
        .filter(df["deadline"].between("2016-01-01","2018-01-01")) \
        .orderBy(desc("deadline"))

dffilter.show()

dffilter.select(sum("pledged")).show()

spark.stop()
```

You can see that this program also prints the sum of the field `pledged`.

Now, try to create your own queries.

3.2 Example: Apache Logs

Contemporary applications and infrastructure software leave behind a tremendous volume of metric and log data. This aggregated "digital exhaust" is inscrutable to humans and difficult for computers to analyze, since it is vast, complex, and not explicitly structured.
This session will introduce the log processing domain and provide practical advice for analyzing log data with Apache Spark.
A publicly available dataset is the NASA Apache Web Logs that have two months of logs. A sample of this dataset that has a single day of logs in TSV (Tab Separated Values) format is also available as nasa_19950801.tsv. In this example it is enough to use this last dataset.
If you're familiar with web servers at all, you'll recognize that this is in Common Log Format. The fields are:

```
remotehost rfc931 authuser [date] "request" status bytes
```

The meaning of these fields is as follows:

field	meaning
remotehost	Remote hostname (or IP number if DNS hostname is not available).
rfc931	The remote logname of the user. We don't really care about this field.
authuser	The username of the remote user, as authenticated by the HTTP server.
[date]	The date and time of the request.
"request"	The request, exactly as it came from the browser or client.
status	The HTTP status code the server sent back to the client.
bytes	The number of bytes (Content-Length) transferred to the client.

In this particular case, the date is represented as a UNIX timestamp and the logname and username have NULL values.
Start by downloading the sample time-series dataset nasa_19950801.tsv.
Then open the Spark console (spark-shell) and experiment with the following commands.

Load the dataset

```
base_df = sqlContext .read .option("header", "true") .option("sep", "\t") .option("inferSchema", "true") .csv("nasa_19950801.tsv")
```

Parse the Unix timestamp column

```
from pyspark.sql.functions import from_unixtime
typed_df = base_df.withColumn("time", from_unixtime("time", "yyyy-MM-dd HH:mm:ss"))
```

Create DataFrame with relevant columns

```
logs_df = typed_df.select("host", "time", "method", "url", "response", "bytes")
```

Check schema and some rows

```
logs_df.printSchema()
logs_df.show()
```

Get statistical parameters (count, mean, stddev, min and max)

```
logs_df.describe(['bytes']).show()
```

Count number of rows per status code

```
status_count_df=logs_df.groupBy('response').count().sort('response').show()
status_count_df.withColumn('log(count)', log(status_count_df['count'])).show()
```

Frequent hosts (> 10 accesses)

```
host_sum_df =(logs_df.groupBy('host').count())
(host_sum_df .filter(host_sum_df['count'] > 10) .select(host_sum_df['host’])) .show(truncate=False)
```

Visualizing paths (URL) from column method

```
paths_df = (logs_df.groupBy('url').count().sort('count', ascending=False))
paths_df.show(truncate=False)
paths_counts = (paths_df .select('url', 'count') .rdd .map(lambda r: (r[0], r[1])) .collect())
for elem in paths_counts : print(elem)
```

Now do the following exercises:

1. **What are the top ten URLs which did not have return code 200?**
   - Create a sorted list containing the URLs and the number of times that they were accessed with a non-200 return code and show the top ten.
2. **How many unique hosts are there in the entire log?**
   - One solution is to use the function `distinct()` upon the select()
3. **Determine the number of unique hosts in the entire log on a day-by-day basis.**
   - We’d like a DataFrame sorted by increasing day of the month which includes the day of the month and the associated number of unique hosts for that day.
4. **Exercise: Determine the average number of requests per host on a day-by-day basis.**

404 errors are returned when the server cannot find the resource (page or object) the browser or client requested. Analyze the occurrence of these errors with the following exercises:

1. **Counting 404 Response Codes**
2. **Listing distinct URLs with 404 Status Code Records**
3. **Listing the Top Twenty URLs with 404 Status Code**
4. **Listing the Top Twenty-five Hosts with 404 Response Code**
5. **Listing 404 Errors per Day**
6. **Top Five Days for 404 Errors**
7. **Hourly 404 Errors**

Now download the NASA Apache Web Logs for one month and repeat the exercises.

4 Using Datasets

The Datasets API provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. You can define a Dataset JVM objects and then manipulate them using functional transformations (`map`, `flatMap`, `filter`, and so on) similar to an RDD. The benefits are that, unlike RDDs, these transformations are now applied on a *structured and strongly typed* distributed collection that allows Spark to leverage Spark SQL's execution engine for optimization.

Convert the final DataFrame from the first Scala example to a DataSet.