DataSet

This tutorial is an introduction to how to load data into Spark.

ratings.csv: _100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users

This Data Set is in CSV format.

SparkSession and Settings

Set up a SparkSession.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("LoadingCSV").getOrCreate()
```

> `RATINGS_CSV_LOCATION` is used to tell our Spark Application where to find the ratings.csv file

Loading Data from a CSV file

`pyspark.sql.DataFrameReader.read.csv` is used for reading csv files.

```
df = spark.read.csv(RATINGS_CSV_LOCATION)
```

You've now told Spark to load the data from the given CSV file. Because Spark is lazy, we have to explicitly tell it to show us something.

Let's see the content by running `.show()` on our new DataFrame.
Let's also check the schema of what we loaded, by using `.printSchema()`.

```
df.show()
df.printSchema()
```

What you can see, is that the data is being loaded, but it does not quite appear to be right. Additionally, all the columns appear to be cast as a StringType - which is not ideal.

Parsing the CSV file correctly and DataTypes()

To parse the CSV correctly, we are going to need to set the following on our `read.csv()` method:

1. We leave the same `path` as before, referring to `RATINGS_CSV_LOCATION` that we set previously.
2. Since we have **comma-seperated-values**, we need to set `sep` to `','`.
3. Since we have a **single header row**, we need to set `header` to `True`.
4. Since columns that contain commas (`,`) are **escaped using double-quotes** (`"`), we set `quote` to `'"'`.
5. Since the files are **encoded as UTF-8**, we set `encoding` to `UTF-8`.
6. Additionally, since we observed that all values are cast to `StringType` by default, we set `inferSchema` to `True`.

```
# Loading CSV file with proper parsing and inferSchema
df = spark.read.csv(
    path=RATINGS_CSV_LOCATION,
    sep=",",
    header=True,
    quote='"',
    encoding="UTF-8",
    inferSchema=True,
)

# Displaying results of the load
df.show()
df.printSchema()
```

Looking at the output we can notice a few things:

- The header now appears properly parsed, no more `_c0`, `_c1`, etc.
- The numeric value columns are cast to `IntegerType` and `DoubleType` thanks to `inferSchema`

Using `inferSchema`, Spark casted the following types to our schema:

> ```
> |-- userId: integer (nullable = true)
> |-- movieId: integer (nullable = true)
> |-- rating: double (nullable = true)
> |-- timestamp: integer (nullable = true)
> ```

In short, our data now appears to have a correct parsed schema with DataTypes that appear to match the current data.

Type Safety

InferSchema is a great way to (quickly) set the schema for the data we are using. It is however good practice to be as explicit as possible when it comes to DataTypes and Schema - we call this Type Safety.

Applying proper schema and ensuring Type Safety, is extra important once we start using more than one Data Source. In this case we will define our `DDL-formatted string` as:

`'userId INT, movieId INT, rating DOUBLE, timestamp INT'`

```
#  Type safe loading of ratings.csv file
df = spark.read.csv(
    path=RATINGS_CSV_LOCATION,
    sep=",",
    header=True,
    quote='"',
    encoding="UTF-8",
    schema="userId INT, movieId INT, rating DOUBLE, timestamp INT",
)

# Displaying results of the load
df.show()
df.printSchema()
df.describe().show()
df.explain()
```

We now have the same output as before, but since we have an explicit schema we can ensure Type Safety

What we've learned so far:

- How to use `read.csv()` to load CSV files, and how to control the settings of this method
- By default, CSVs are parsed with all columns being cast to `StringType`
- `inferSchema` allows Spark to guess what schema should be used
- To ensure proper Type Safety, we can use Hive Schema DDL to set an explicit schema

```
spark.stop()
```