Lab 4.4: 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 Zeppelin 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.

Python users should **skip this lab**

Builds on

Lab 4.3

Run time

30-40 minutes

Music Data File

We've provided a small and simple JSON data file (***spark-labs/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
- `category`: The music category (e.g. Pop)

Tasks

- Create a DataFrame by reading in the music data in ***data/music.json***.

```
> val musicDF = //
```

- 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.
- Using this class, create a typed Dataset from the `musicDF` dataframe created with this data in the previous lab.

```
> case class MusicItem ...
> val musicDS = // ...
```

- What type is `musicDS` ? Display
- the data in the dataset.

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?

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 4.5: 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. For more info on the data itself, see the notes at the end of the lab.

Builds on

None

Run time

30-40 minutes

Wikimedia PageView Data File

We provide a data file (***spark-labs/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.

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

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

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 funtions. We've used this in our word count examples in the main manual.

Tasks

- Create a dataframe by splitting each line up.
  - 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
  - Call the resulting column `splitLineDF`
  - To rename a column, use **Scala:**
    - `as(newName)` **Python:** `alias(newName)`

```
// Scala
> val splitViewsDF = // ...
```

```
# Python
# import everyting in the functions module
> from pyspark.sql.functions import *
> splitViewsDF = // ...
```

- View the schema of this DataFrame.
- Display a few rows in the DataFrame (there are many).
  - If the display of a row is truncated, you can pass an argument to `show()` to prevent truncation.
  - **Scal:a** `show(false)`

    - **Python:** `show(truncate=False)`
- This is slightly better, but still unwieldy. We'll
- apply a finer grained schema next.

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.

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

Tasks

- Use a select to create a dataframe with the schema above.
    - You can select the nth element of a row containing an array via syntax like this:
        - **Scala:** `splitLine(n)`
        - **Python:** `splitLine[n]`
    - You can cast an element using `cast(dataType)`.
        - Providing data type to convert to, e.g. "integer", "long", "boolean", etc.
    - You can name a selected column using:
        - **Scala:** `as(newName)`
        - **Python:** `alias(newName)`
- Once you've created the dataframe, then: View
    - the schema.
    - View a few rows of the data.
    - Your data should look something like the below.

```
+------+--------+---------+----+
|domain| pageName|viewCount|size|
+------+--------+---------+----+
|    aa|Main_Page|        3|   0|
|    aa|Main_page|        1|   0|
|    aa|User:Savh|        1|   0|
|    aa|Wikipedia|        1|   0|
|  aa.b|User:Savh|        1|   0|
+------+--------+---------+----+
```

Try Some Queries

Tasks

Try some queries on the dataframe you've created, e.g.

- Find rows where the viewCount > 500, and display 5 of them.
  - Do the same query as above, but also filter for where the domain is "en".
- Find the 5 rows with the largest viewCount and a domain of "en".
  - **Python:** When ordering with `orderBy()` , you can use either of the following
    - `orderBy(column.asc())` / `orderBy(colum.desc())`
    - `orderBy("columName", ascending=True)` /
    - `orderBy("columName", ascending=False)`

Optional Tasks

Try some other queries, e.g.

- Find the 5 rows with the largest viewCount and a domain of "en", and where the pageName doesn't contain a colon (":").
  - To check if a column contains a string use:
    - **Scala:** `yourColumn.contains("string_to_match")`
    - **Python:** `yourColumn.like("%string_to_match%")`
  - To negate in a DataFrame boolean expression, you use
    - **Scala:** `!` (exclamation)
    - **Python:** `~` (tilde)
- **[Optional]** Think up your own and try them out.

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 di"cult, then the full power of DataFrames can be easily applied.

Notes on Data File

The data file we use was downloaded from: Wikimedia March data. For

more information on Wikimedia data, you can see these links:

- Pageview API
- Field Description - this is for the older (deprecated) Pagecount API, but describes the dump files from the current API also.