BIG DATA ANALYTICS

Spark Datasets

Spark Datasets: *Outline*

- Introduction 
- When to Use Datasets
- Creating Datasets
- Actions
- Transformations

Spark Datasets: *Introduction*

- Datasets are a core part of Spark's Structured APIs, specifically available in Scala and Java (JVM languages).
- They allow us to define the object type for each row, making Datasets a "typed" version of the API.

Spark Datasets: *Introduction*

Key Points:

- DataFrames are actually Datasets of type `Row`. Datasets enable more control over the schema by using **`Scala case classes`** or **`JavaBeans`**.
- Encoders are used to convert domain-specific objects (like a `Person` class with `name` and `age` fields) into Spark's internal binary format. This enables Spark to handle our custom objects in distributed computations.
- Performance: Using Datasets can slightly slow down operations due to the conversion between custom objects and Spark's `Row` format. However, this is more efficient than using Python UDFs, as the overhead is less significant.

Spark Datasets: *Introduction*

Example:

- In Scala, a case class can be used to define a `Person` object:

```
// scala

case class Person(name: String, age: Int)

val peopleDS = Seq(Person("John", 30)).toDS()
```

- While Datasets provide more flexibility, be mindful of potential performance impacts when working with domain-specific objects.

Spark Datasets: *Outline*

- Introduction
- When to Use Datasets
- Creating Datasets
- Actions
- Transformations

Spark Datasets: When to Use Datasets

Although Datasets may come with a performance cost, there are specific scenarios where they are beneficial. Here are the main reasons to use Datasets:

1. When DataFrame operations are insufficient: Some complex operations can't be easily expressed using DataFrame manipulations or SQL. Datasets allow us to encode complex business logic directly in our code.
2. When type safety is important: Datasets offer type-safety, meaning errors (like subtracting strings) are caught at compile-time, not runtime. This is useful for ensuring code correctness, though it comes at a performance cost.
3. Reusing transformations: Datasets are similar to Scala Sequence Types but operate in a distributed way. We can reuse transformations between local and distributed workloads easily by defining case classes in Scala.

Spark Datasets: When to Use Datasets

4. Combining DataFrames and Datasets: A common approach is to use DataFrames for performance and switch to Datasets when type-safety or row-level transformations are needed. For example, use DataFrames for most ETL (Extract, transform, and load) work, and then switch to Datasets for more detailed manipulation or validation.

This hybrid approach allows us to balance between performance and safety, depending on the stage of our workflow.

Spark Datasets: *Outline*

- Introduction
- When to Use Datasets
- Creating Datasets
- Actions
- Transformations

Spark Datasets: Creating Datasets

Creating Datasets is somewhat of a manual operation, requiring us to know and define the schemas ahead of time.

In Java: Encoders

Java Encoders are fairly simple, we simply specify our class and then we'll encode it when we come upon our DataFrame (which is of type `Dataset<Row>`):

```
import org.apache.spark.sql.Encoders;
public class Flight implements Serializable{
  String DEST_COUNTRY_NAME;
  String ORIGIN_COUNTRY_NAME;
  Long DEST_COUNTRY_NAME;
}
Dataset<Flight> flights = spark.read
  .parquet("/data/flight-data/parquet/2010-summary.parquet/")
  .as(Encoders.bean(Flight.class));
```

Spark Datasets: Creating Datasets

In Scala: Case Classes

To create Datasets in Scala, we define a Scala `case class`. A `case class` is a regular class that has the following characteristics:

- Immutable
- Decomposable through pattern matching
- Allows for comparison based on structure instead of reference
- Easy to use and manipulate

These traits make it rather valuable for data analysis because it is quite easy to reason about a case class.

Probably the most important feature is that case classes are immutable and allow for comparison by structure instead of value.

Spark Datasets: Creating Datasets

In Scala: Case Classes

Here's how the Scala documentation describes it:

- Immutability frees us from needing to keep track of where and when things are mutated
- Comparison-by-value allows us to compare instances as if they were primitive values-no more uncertainty regarding whether instances of a class are compared by value or reference
- Pattern matching simplifies branching logic, which leads to less bugs and more readable code.

These advantages carry over to their usage within Spark, as well.

Spark Datasets: Creating Datasets

In Scala: Case Classes

To begin creating a Dataset, let's define a `case class` for one of our datasets:

```
case class Flight(DEST_COUNTRY_NAME: String,
                  ORIGIN_COUNTRY_NAME: String, count: BigInt)
```

Now that we defined a `case class`, this will represent a single record in our dataset. More concisely, we now have a Dataset of Flights. This doesn't define any methods for us, simply the schema. When we read in our data, we'll get a DataFrame. However, we simply use the as method to cast it to our specified row type:

```
val flightsDF = spark.read
.parquet("/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
```

Spark Datasets: *Outline*

- Introduction
- When to Use Datasets
- Creating Datasets
- Actions
- Transformations

Spark Datasets: Actions

Even though we can see the power of Datasets, what's important to understand is that actions like `collect`, `take`, and `count` apply to whether we are using Datasets or DataFrames:

```
flights.show(2)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
|    United States|            Ireland|  264|
+-----------------+-------------------+-----+
```

Spark Datasets: Actions

We'll also notice that when we actually go to access one of the `case classes`, we don't need to do any type coercion, we simply specify the named attribute of the `case class` and get back, not just the expected value but the expected type, as well:

```
flights.first.DEST_COUNTRY_NAME // United States
```

Spark Datasets: *Outline*

- Introduction
- When to Use Datasets
- Creating Datasets
- Actions
- Transformations

Spark Datasets: Transformations

Transformations on Datasets and DataFrames:

- Datasets support the same transformations as DataFrames, including joins and aggregations.
- Datasets allow for more complex, strongly-typed transformations because they work with raw JVM types.
- Example: We can filter a Dataset using raw object manipulation.

Spark Datasets: Transformations

1. Filtering:

Filtering Example

- Create a simple function to check if the origin and destination are the same:

```
// scala
def originIsDestination(flight_row: Flight): Boolean = {
    return flight_row.ORIGIN_COUNTRY_NAME ==
flight_row.DEST_COUNTRY_NAME
}
```

Spark Datasets: Transformations

1. Filtering: cont'd.

Filtering Example cont'd.

- Use this function to filter the Dataset:

```
// scala
  flights.filter(flight_row =>
originIsDestination(flight_row)).first()
```

Result:

```
  Flight(United States, United States, 348113)
```

- This function can be tested locally before using it in Spark:

```
  // scala
  flights.collect().filter(flight_row =>
originIsDestination(flight_row))
```

Result:

```
Array(Flight(United States, United States, 348113))
```

Spark Datasets: Transformations

2. Mapping:

Mapping Example

- Mapping transforms values from one type to another.
- Example: Extract the destination country from each flight:

```
// scala
val destinations = flights.map(f => f.DEST_COUNTRY_NAME)
```

Result: A Dataset of type `String` containing destination country names.

- Collect the first 5 destinations as an array:

```
// scala
val localDestinations = destinations.take(5)
```

- This is similar to a `select` in DataFrames, but with compile-time checking and is useful for more complex row-by-row operations.