**Chapter 11. Datasets**

Datasets are the foundational type of the Structured APIs. We already worked with DataFrames, which are Datasets of type Row, and are available across Spark’s different languages. Datasets are a strictly Java Virtual Machine (JVM) language feature that work only with Scala and Java. Using Datasets, you can define the object that each row in your Dataset will consist of. In Scala, this will be a case class object that essentially defines a schema that you can use, and in Java, you will define a Java Bean. Experienced users often refer to Datasets as the “typed set of APIs” in Spark. For more information, see [Chapter 4](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch04.html#s2c1---structured-api-overview).

In [Chapter 4](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch04.html#s2c1---structured-api-overview), we discussed that Spark has types like StringType, BigIntType, StructType, and so on. Those Spark-specific types map to types available in each of Spark’s languages like String, Integer, and Double. When you use the DataFrame API, you do not create strings or integers, but Spark manipulates the data for you by manipulating the Row object. In fact, if you use Scala or Java, all “DataFrames” are actually Datasets of type Row. To efficiently support domain-specific objects, a special concept called an “Encoder” is required. The encoder maps the domain-specific type T to Spark’s internal type system.

For example, given a class Person with two fields, name (string) and age (int), an encoder directs Spark to generate code at runtime to serialize the Person object into a binary structure. When using DataFrames or the “standard” Structured APIs, this binary structure will be a Row. When we want to create our own domain-specific objects, we specify a case class in Scala or a JavaBean in Java. Spark will allow us to manipulate this object (in place of a Row) in a distributed manner.

When you use the Dataset API, for every row it touches, this domain specifies type, Spark converts the Spark Row format to the object you specified (a case class or Java class). This conversion slows down your operations but can provide more flexibility. You will notice a hit in performance but this is a far different order of magnitude from what you might see from something like a user-defined function (UDF) in Python, because the performance costs are not as extreme as switching programming languages, but it is an important thing to keep in mind.

**When to Use Datasets**

You might ponder, if I am going to pay a performance penalty when I use Datasets, why should I use them at all? If we had to condense this down into a canonical list, here are a couple of reasons:

* When the operation(s) you would like to perform cannot be expressed using DataFrame manipulations
* When you want or need type-safety, and you’re willing to accept the cost of performance to achieve it

Let’s explore these in more detail. There are some operations that cannot be expressed using the Structured APIs we have seen in the previous chapters. Although these are not particularly common, you might have a large set of business logic that you’d like to encode in one specific function instead of in SQL or DataFrames. This is an appropriate use for Datasets. Additionally, the Dataset API is type-safe. Operations that are not valid for their types, say subtracting two string types, will fail at compilation time not at runtime. If correctness and bulletproof code is your highest priority, at the cost of some performance, this can be a great choice for you. This does not protect you from malformed data but can allow you to more elegantly handle and organize it.

Another potential time for which you might want to use Datasets is when you would like to reuse a variety of transformations of entire rows between single-node workloads and Spark workloads. If you have some experience with Scala, you might notice that Spark’s APIs reflect those of Scala Sequence Types, but they operate in a distributed fashion. In fact, Martin Odersky, the inventor of Scala, said [just that in 2015 at Spark Summit Europe](https://spark-summit.org/eu-2015/events/spark-the-ultimate-scala-collections/). Due to this, one advantage of using Datasets is that if you define all of your data and transformations as accepting case classes it is trivial to reuse them for both distributed and local workloads. Additionally, when you collect your DataFrames to local disk, they will be of the correct class and type, sometimes making further manipulation easier.

Probably the most popular use case is to use DataFrames and Datasets in tandem, manually trading off between performance and type safety when it is most relevant for your workload. This might be at the end of a large, DataFrame-based extract, transform, and load (ETL) transformation when you’d like to collect data to the driver and manipulate it by using single-node libraries, or it might be at the beginning of a transformation when you need to perform per-row parsing before performing filtering and further manipulation in Spark SQL.

**Creating Datasets**

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

**In Java: Encoders**

Java Encoders are fairly simple, you simply specify your class and then you’ll encode it when you come upon your 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));

**In Scala: Case Classes**

To create Datasets in Scala, you 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.

Here’s how [the Scala documentation](http://docs.scala-lang.org/tutorials/tour/case-classes.html) describes it:

* Immutability frees you from needing to keep track of where and when things are mutated
* Comparison-by-value allows you 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.

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 succintly, 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**]

**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|

+-----------------+-------------------+-----+

You’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*

**Transformations**

Transformations on Datasets are the same as those that we saw on DataFrames. Any transformation that you read about in this section is valid on a Dataset, and we encourage you to look through the specific sections on relevant aggregations or joins.

In addition to those transformations, Datasets allow us to specify more complex and strongly typed transformations than we could perform on DataFrames alone because we manipulate raw Java Virtual Machine (JVM) types. To illustrate this raw object manipulation, let’s filter the Dataset that you just created.

**Filtering**

Let’s look at a simple example by creating a simple function that accepts a Flight and returns a Boolean value that describes whether the origin and destination are the same. This is not a UDF (at least, in the way that Spark SQL defines UDF) but a generic function.

**TIP**

You’ll notice in the following example that we’re going to create a *function* to define this filter. This is an important difference from what we have done thus far in the book. By specifying a function, we are *forcing* Spark to evaluate this function on every row in our Dataset. This can be very resource intensive. For simple filters it is always preferred to write SQL expressions. This will greatly reduce the cost of filtering out the data while still allowing you to manipulate it as a Dataset later on:

**def** originIsDestination(flight\_row**:** **Flight**)**:** **Boolean** = {

**return** flight\_row.**ORIGIN\_COUNTRY\_NAME** == flight\_row.**DEST\_COUNTRY\_NAME**

}

We can now pass this function into the filter method specifying that for each row it should verify that this function returns true and in the process will filter our Dataset down accordingly:

flights.filter(flight\_row **=>** originIsDestination(flight\_row)).first()

The result is:

Flight = Flight(United States,United States,348113)

As we saw earlier, this function does not need to execute in Spark code at all. Similar to our UDFs, we can use it and test it on data on our local machines before using it within Spark.

For example, this dataset is small enough for us to collect to the driver (as an Array of Flights) on which we can operate and perform the exact same filtering operation:

flights.collect().filter(flight\_row **=>** originIsDestination(flight\_row))

The result is:

Array[Flight] = Array(Flight(United States,United States,348113))

We can see that we get the exact same answer as before.

**Mapping**

Filtering is a simple transformation, but sometimes you need to map one value to another value. We did this with our function in the previous example: it accepts a flight and returns a Boolean, but other times we might actually need to perform something more sophisticated like extract a value, compare a set of values, or something similar.

The simplest example is manipulating our Dataset such that we extract one value from each row. This is effectively performing a DataFrame like select on our Dataset. Let’s extract the destination:

**val** destinations **=** flights.map(f **=>** f.**DEST\_COUNTRY\_NAME**)

Notice that we end up with a Dataset of type String. That is because Spark already knows the JVM type that this result should return and allows us to benefit from compile-time checking if, for some reason, it is invalid.

We can collect this and get back an array of strings on the driver:

**val** localDestinations **=** destinations.take(5)

This might feel trivial and unnecessary; we can do the majority of this right on DataFrames. We in fact recommend that you do this because you gain so many benefits from doing so. You will gain advantages like code generation that are simply not possible with arbitrary user-defined functions. However, this can come in handy with much more sophisticated row-by-row manipulation.

**Joins**

Joins, as we covered earlier, apply just the same as they did for DataFrames. However Datasets also provide a more sophisticated method, the joinWith method. joinWith is roughly equal to a co-group (in RDD terminology) and you basically end up with two nested Datasets inside of one. Each column represents one Dataset and these can be manipulated accordingly. This can be useful when you need to maintain more information in the join or perform some more sophisticated manipulation on the entire result, like an advanced map or filter.

Let’s create a fake flight metadata dataset to demonstrate joinWith:

**case** **class** **FlightMetadata**(count**:** **BigInt**, randomData**:** **BigInt**)

**val** flightsMeta **=** spark.range(500).map(x **=>** (x, scala.util.**Random**.nextLong))

.withColumnRenamed("\_1", "count").withColumnRenamed("\_2", "randomData")

.as[**FlightMetadata**]

**val** flights2 **=** flights

.joinWith(flightsMeta, flights.col("count") === flightsMeta.col("count"))

Notice that we end up with a Dataset of a sort of key-value pair, in which each row represents a Flight and the Flight Metadata. We can, of course, query these as a Dataset or a DataFrame with complex types:

flights2.selectExpr("\_1.DEST\_COUNTRY\_NAME")

We can collect them just as we did before:

flights2.take(2)

Array[(Flight, FlightMetadata)] = Array((Flight(United States,Romania,1),...

Of course, a “regular” join would work quite well, too, although you’ll notice in this case that we end up with a DataFrame (and thus lose our JVM type information).

**val** flights2 **=** flights.join(flightsMeta, **Seq**("count"))

We can always define another Dataset to gain this back. It’s also important to note that there are no problems joining a DataFrame and a Dataset—we end up with the same result:

**val** flights2 **=** flights.join(flightsMeta.toDF(), **Seq**("count"))

**Grouping and Aggregations**

Grouping and aggregations follow the same fundamental standards that we saw in the previous aggregation chapter, so groupBy rollup and cube still apply, but these return DataFrames instead of Datasets (you lose type information):

flights.groupBy("DEST\_COUNTRY\_NAME").count()

This often is not too big of a deal, but if you want to keep type information around there are other groupings and aggregations that you can perform. An excellent example is the groupByKey method. This allows you to group by a specific key in the Dataset and get a typed Dataset in return. This function, however, doesn’t accept a specific column name but rather a function. This makes it possible for you to specify more sophisticated grouping functions that are much more akin to something like this:

flights.groupByKey(x **=>** x.**DEST\_COUNTRY\_NAME**).count()

Although this provides flexibility, it’s a trade-off because now we are introducing JVM types as well as functions that cannot be optimized by Spark. This means that you will see a performance difference and we can see this when we inspect the explain plan. In the following, you can see that we are effectivelly appending a new column to the DataFrame (the result of our function) and then performing the grouping on that:

flights.groupByKey(x **=>** x.**DEST\_COUNTRY\_NAME**).count().explain

== Physical Plan ==

\*HashAggregate(keys=[value#1396], functions=[count(1)])

+- Exchange hashpartitioning(value#1396, 200)

+- \*HashAggregate(keys=[value#1396], functions=[partial\_count(1)])

+- \*Project [value#1396]

+- AppendColumns <function1>, newInstance(class ...

[staticinvoke(class org.apache.spark.unsafe.types.UTF8String, ...

+- \*FileScan parquet [D...

After we perform a grouping with a key on a Dataset, we can operate on the Key Value Dataset with functions that will manipulate the groupings as raw objects:

**def** grpSum(countryName**:String**, values**:** **Iterator**[**Flight**]) **=** {

values.dropWhile(**\_**.count < 5).map(x **=>** (countryName, x))

}

flights.groupByKey(x **=>** x.**DEST\_COUNTRY\_NAME**).flatMapGroups(grpSum).show(5)

+--------+--------------------+

| \_1| \_2|

+--------+--------------------+

|Anguilla|[Anguilla,United ...|

|Paraguay|[Paraguay,United ...|

| Russia|[Russia,United St...|

| Senegal|[Senegal,United S...|

| Sweden|[Sweden,United St...|

+--------+--------------------+

**def** grpSum2(f**:Flight**)**:Integer** = {

1

}

flights.groupByKey(x **=>** x.**DEST\_COUNTRY\_NAME**).mapValues(grpSum2).count().take(5)

We can even create new manipulations and define how groups should be reduced:

**def** sum2(left**:Flight**, right**:Flight**) **=** {

**Flight**(left.**DEST\_COUNTRY\_NAME**, **null**, left.count + right.count)

}

flights.groupByKey(x **=>** x.**DEST\_COUNTRY\_NAME**).reduceGroups((l, r) **=>** sum2(l, r))

.take(5)

It should be straightfoward enough to understand that this is a more expensive process than aggregating immediately after scanning, especially because it ends up in the same end result:

flights.groupBy("DEST\_COUNTRY\_NAME").count().explain

== Physical Plan ==

\*HashAggregate(keys=[DEST\_COUNTRY\_NAME#1308], functions=[count(1)])

+- Exchange hashpartitioning(DEST\_COUNTRY\_NAME#1308, 200)

+- \*HashAggregate(keys=[DEST\_COUNTRY\_NAME#1308], functions=[partial\_count(1)])

+- \*FileScan parquet [DEST\_COUNTRY\_NAME#1308] Batched: tru...

This should motivate using Datasets only with user-defined encoding surgically and only where it makes sense. This might be at the beginning of a big data pipeline or at the end of one.

**Conclusion**

In this chapter, we covered the basics of Datasets and provided some motivating examples. Although short, this chapter actually teaches you basically all that you need to know about Datasets and how to use them. It can be helpful to think of them as a blend between the higher-level Structured APIs and the low-level RDD APIs, which is the topic of [Chapter 12](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch12.html#s3c1---rdd-basics).