Ingesting an XML file

In this section, you will ingest an *Extensible Markup Language* (*XML*) document containing National Aeronautics and Space Administration (NASA) patents and then display some patents and the dataframe's schema. Note that, in this context, the schema is not an XML Schema (or XSD), but the dataframe schema. Quite a few years ago, when I discovered XML, I really thought it could become a unified lingua franca of data exchange. XML can be described as follows:

- Structured
- Extensible
- Self-describing
- Embeds validation rules through document type definition (DTD) and XML Schema Definition (XSD)
- A W3 standard

You can read more about XML at https://www.w3.org/XML/. XML looks like HTML and any other markup language since SGML:

```
<rootElement>
  <element attribute="attribute's value">
    Some payload in a text element
  </element>
  <element type="without sub nodes"/>
</rootElement>
```

Unfortunately, XML is verbose and harder to read than JSON. Nevertheless, XML is still widely used, and Apache Spark ingests it nicely.

Figure 7.6 shows a fragment of the XML file and illustrates the process.

For this XML example, you are going to ingest the NASA patents. NASA offers various open datasets at https://data.nasa.gov. Listing 7.13 shows a record of this file.

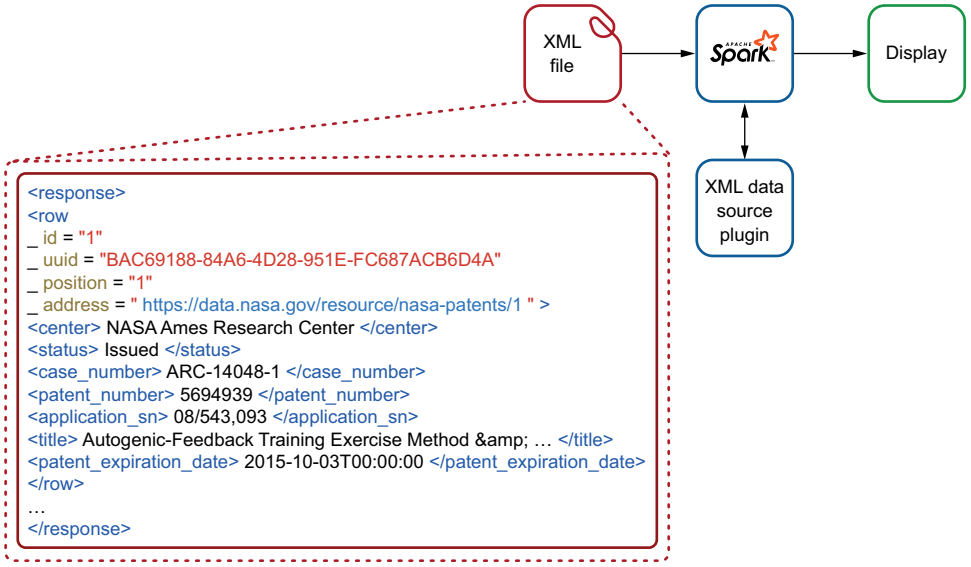

Figure 7.6 Spark ingests an XML file containing NASA patents. Spark uses an external plugin, provided by Databricks, to perform the ingestion. Spark will then display records and the dataframe schema (not to be confused with an XML Schema).

LAB You can download the NASA patents dataset from https://data.nasa.gov/Raw-Data/NASA-Patents/gquh-watm. For this example, I used Spark v2.2.0 on Mac OS X v10.12.6 with Java 8 as well as the Databricks XML parser v0.4.1. The dataset was downloaded in January 2018.

Listing 7.13 NASA patents (excerpt)

Root element of your list of patents

Element (or tag) designing our record

Attributes are prefixed by one underscore (_).

```
<response>
  <row
      _id="1"
      _uuid="BAC69188-84A6-4D28-951E-FC687ACB6D4A"
      _position="1"
      _address="https://data.nasa.gov/resource/nasa-patents/1">
    <center>NASA Ames Research Center</center>
    <status>Issued</status>
    <case_number>ARC-14048-1</case_number>
    <patent_number>5694939</patent_number>
    <application_sn>08/543,093</application_sn>
    <title>Autogenic-Feedback Training Exercise Method &amp;amp; System</title>
    <patent_expiration_date>2015-10-03T00:00:00</patent_expiration_date>
  </row>
…
</response>
```

Desired output

Listing 7.14 shows the output of a dataframe's data and schema after ingesting the NASA patents as an XML document. You can see that the attributes are prefixed by an underscore (attributes already had an underscore as a prefix in the original document, so now they have two), and the element's name is used as a column name.

Listing 7.14 NASA patents in a dataframe

```
+--------------------+----+----------+--------------------+--------------+…
|           __address|__id|__position|              __uuid|application_sn|…
+--------------------+----+----------+--------------------+--------------+…
|https://data.nasa...| 407|       407|2311F785-C00F-422...|    13/033,085|…
|https://data.nasa...|   1|         1|BAC69188-84A6-4D2...|    08/543,093|…
|https://data.nasa...|   2|         2|23D6A5BD-26E2-42D...|    09/017,519|…
|https://data.nasa...|   3|         3|F8052701-E520-43A...|    10/874,003|…
|https://data.nasa...|   4|         4|20A4C4A9-EEB6-45D...|    09/652,299|…
+--------------------+----+----------+--------------------+--------------+…
only showing top 5 rows

root
 |-- __address: string (nullable = true)
 |-- __id: long (nullable = true)
 |-- __position: long (nullable = true)
 |-- __uuid: string (nullable = true)
 |-- application_sn: string (nullable = true)
 |-- case_number: string (nullable = true)
 |-- center: string (nullable = true)
 |-- patent_expiration_date: string (nullable = true)
 |-- patent_number: string (nullable = true)
 |-- status: string (nullable = true)
 |-- title: string (nullable = true)
```

Code

As usual, our code will start with a `main()` method, which calls a `start()` method to create a Spark session. The following listing is the Java code needed to ingest the NASA XML file and then display five records and its schema.

Listing 7.15 XmlToDataframeApp.java

```
package net.jgp.books.spark.ch07.lab600_xml_ingestion;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;

public class XmlToDataframeApp {

  public static void main(String[] args) {
    XmlToDataframeApp app = new XmlToDataframeApp();
    app.start();
```

```
    }

    private void start() {
      SparkSession spark = SparkSession.builder()
          .appName("XML to Dataframe")
          .master("local")
          .getOrCreate();

      Dataset<Row> df = spark.read().format("xml")
          .option("rowTag", "row")
          .load("data/nasa-patents.xml");

      df.show(5);
      df.printSchema();
    }
}
```

Element or tag that indicates a record in the XML file

Specifies XML as the format. Case does not matter.

I had to modify the original NASA document because it contained an element with the same name as the record, wrapping the records. Unfortunately, as of now, Spark cannot change this element's name for us. The original structure was as follows:

```
<response>
  <row>
    <row _id="1" …>
      …
    </row>
    …
  </row>
</response>
```

If the first child of `response` had been `rows`, or anything else other than `row`, I wouldn't have had to remove it (another option is to rename it).

Because the parser is not part of the standard Spark distribution, you will have to add it to the pom.xml file, as described in the following listing. To ingest XML, you will use spark-xml_2.12 (the artifact) from Databricks, in version 0.7.0.

Listing 7.16 pom.xml to ingest XML (excerpt)

```
…
  <properties>
    …
    <scala.version>2.12</scala.version>
    <spark-xml.version>0.7.0</spark-xml.version>
  </properties>

  <dependencies>
…
    <dependency>
      <groupId>com.databricks</groupId>
      <artifactId>spark-xml_${scala.version}</artifactId>
      <version>${spark-xml.version}</version>
      <exclusions>
```

Scala version on which the XML is built

Version of the XML parser

Equivalent to spark-xml_2.12

Optional: I excluded the logger from other packages to have better control over the one I use.

Equivalent to 0.7.0

```
        <exclusion>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-simple</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
…
  </dependencies>
…
```

More details on Spark XML can be found at https://github.com/databricks/spark-xml.

70

First of all, Spark only starts reading in the data when an action (like `count`, `collect` or `write`) is called. Once an action is called, Spark loads in data in *partitions* - the number of concurrently loaded partitions depend on the number of cores you have available. So in Spark you can think of 1 partition = 1 core = 1 task. Note that all concurrently loaded partitions have to fit into memory, or you will get an OOM.

Assuming that you have several stages, Spark then runs the transformations from the first stage on the loaded partitions only. Once it has applied the transformations on the data in the loaded partitions, it stores the output as shuffle-data and then reads in more partitions. It then applies the transformations on these partitions, stores the output as shuffle-data, reads in more partitions and so forth until all data has been read.

If you apply no transformation but only do for instance a `count`, Spark will still read in the data in partitions, but it will not store any data in your cluster and if you do the `count` again it will read in all the data once again. To avoid reading in data several times, you might call `cache` or `persist` in which case Spark *will* try to store the data in you cluster. On `cache` (which is the same as `persist(StorageLevel.MEMORY_ONLY)` it will store all partitions in memory - if it doesn't fit in memory you will get an OOM. If you call `persist(StorageLevel.MEMORY_AND_DISK)` it will store as much as it can in memory and the rest will be put on disk. If data doesn't fit on disk either the OS will usually kill your workers.

Note that Spark has its own little memory management system. Some of the memory that you assign to your Spark job is used to hold the data being worked on and some of the memory is used for storage if you call `cache` or `persist`.

I hope this explanation helps :)

Some useful information i found on Stack overflow regarding partitions