You can find the sample CSV and Excel files that are used in this section within my GitHub Repo here: https://github.com/ronlesteve/the-azure-data-lakehouse-toolkit/tree/main/ExcelFiles.

Here is the PySpark code that you will need to run to recreate the results shown in Figure 6-2.

excel=spark.read.format("com.crealytics.spark.excel")\

.option("header", "true")\

.option("sheetName", "Orders")\

.option("dataAddress", "'Orders'!A1:F600")\

.option("inferSchema", "true")\

.option("useHeader", "true")\

.option("treatEmptyValuesAsNulls", "true")\

.option("addColorColumns", "False")\

.load("/mnt/raw/Orders.xlsx")

The following PySpark code shows how to read a CSV file and load it to a dataframe. With this method, there is no need to refer to the Spark Excel Maven Library in the code.

csv=spark.read.format("csv")\

.option("header", "true")\

.option("inferSchema", "true")\

.load("/mnt/raw/dimdates.csv")

Notice from the sample PySpark code below that you have the option to select the content from a CSV file and write it to an Excel file with the help of the Spark Excel Maven library.

csv.select("\*").write.format('com.crealytics.spark.excel')\

.option("header","true")\

.option("inferSchema","true")\

.save('/mnt/raw/dimdate.xlsx',mode="overwrite")

Run the following PySpark code to read the Excel file into a dataframe.

csv\_to\_xls=spark.read.format("com.crealytics.spark.excel")\

.option("header", "true")\

.option("inferSchema", "true")\

.load("/mnt/raw/dimdate.xlsx")

You can then write custom PySpark code to extract, transform and load data within your Excel file.

import xlsxwriter

from shutil import copyfile

workbook = xlsxwriter.Workbook("/mnt/raw/Orders.xlsx")

worksheet = workbook.add\_worksheet()

worksheet.write(0, 0, "Key")

worksheet.write(0, 1, "Value")

workbook.close()

copyfile('/mnt/raw/Orders.xlsx', '/mnt/raw/dimdate.xlsx')

The code below shows a few sample records of the XML file books.xml that is used in this example. This is a Microsoft sample file which can be found here: <https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v=vs.85)>. In addition, you can find another sample xml file related to Purchase Orders here: https://docs.microsoft.com/en-us/dotnet/standard/linq/sample-xml-file-multiple-purchase-orders.

<?xml version="1.0"?>

<catalog>

<book id="bk101">

<author>LEsteve, Ron</author>

<title>The Definitive Guide to Azure Data Engineering</title>

<genre>Computer</genre>

<price>44.95</price>

<publish\_date>2020-08-01</publish\_date>

<description>Modern ELT, DevOps, and Analytics on the Azure Cloud Platform.</description>

</book>

<book id="bk102">

<author>Ralls, Kim</author>

<title>Midnight Rain</title>

<genre>Fantasy</genre>

<price>5.95</price>

<publish\_date>2000-12-16</publish\_date>

<description>A former architect battles corporate zombies,

an evil sorceress, and her own childhood to become queen

of the world.</description>

</book>

<book id="bk103">

<author>Corets, Eva</author>

<title>Maeve Ascendant</title>

<genre>Fantasy</genre>

<price>5.95</price>

<publish\_date>2000-11-17</publish\_date>

<description>After the collapse of a nanotechnology

society in England, the young survivors lay the

foundation for a new society.</description>

</book>

</catalog>

Here is the PySpark code that you will need to run to recreate the results shown in Figure 6-9.

df=spark.read.text("/mnt/raw/books.xml")

display(df)

Here is the PySpark code that you will need to run to recreate the results shown in Figure 6-10. rowTag is the row tag to treat as a row and rootTag is the root tag to treat as the root.

df=spark.read.format("com.databricks.spark.xml")\

.option("rootTag", "Catalog")\

.option("rowTag","book")\

.load("/mnt/raw/books.xml")

display(df)

With this next block of PySpark code, you will be able to use the spark xml package to write the results of the dataframe back to an xml file called booksnew.xml.

df.select("\*").write.format('com.databricks.spark.xml')\

.option("rootTag", "Catalog")\

.option("rowTag","book")\

.save('/mnt/raw/booksnew.xml',mode="overwrite")

You can then run the following code to read the file and retrieve the results into a dataframe.

df=spark.read.format("com.databricks.spark.xml")\

.option("rootTag", "Catalog")\

.option("rowTag","book")\

.load("/mnt/raw/booksnew.xml")

display(df)

Finally, you could also create a SQL table using the following syntax which specifies the xml format, xml file path, and rowTag. With this table created, you’ll be able to write SQL scripts to query your xml data in tabular format.

%sql

CREATE TABLE books

USING xml

OPTIONS (path "/mnt/raw/books.xml", rowTag "book");

SELECT \* FROM books;

For reference, here are the first three rows of the Customer1 file to shown the structure of the json format.

{"customerid":1,"Title":"Mr.","FirstName":"Orlando","MiddleName":"N.","LastName":"Gee"}

{"customerid":2,"Title":"Mr.","FirstName":"Keith","LastName":"Harris"}

{"customerid":3,"Title":"Ms.","FirstName":"Donna","MiddleName":"F.","LastName":"Carreras"}

This next block of code is SQL syntax which can also be run within your Python notebook by specifying the %sql command in the beginning of the script. With the following scripts, you will be able to create temporary SQL view of the json format data. You could then write SQL statements to query the view just as you would a regular SQL table to retrieve the results in tabular format.

%sql

CREATE TEMPORARY VIEW json\_table

USING json

OPTIONS (path "/mnt/raw/Customer1.json")

%sql

SELECT \* FROM json\_table

WHERE customerid>5

In the next scenario, you can read multiline json data using simple PySpark commands. First, you’ll need to create a json file containing multiline data, as shown in the code below. This code will create a multiline.json file within your mounted ADLSgen2 account.

dbutils.fs.put("/mnt/raw/multiline.json", """

[

{"string":"string1","int":1,"array":[0,1,2],"key/value": {"key": "value1"}},

{"string":"string2","int":2,"array":[3,4,5],"key/value": {"key": "value2"}},

{"string":"string2","int":2,"array":[6,7,8],"key/value": {"key": "value2"}}

]""",True)

After the file is created, you can read the file by running the following script: multiline\_json=spark.read.option('multiline',"true").json("/mnt/raw/multiline.json"). After that, the display(multiline\_json) command will retrieve the multi-line json data with the capability of expanding the data within each row, as shown in Figure 6-12.

In the code shown below, you would store the JSON object per string in a dataframe, create an RDD using the sc.parallelize command, and finally you’ll be able to read the data

# RDD[String] storing a JSON object per string

data = ['{"booktitle":"The Azure Data Lakehouse Toolkit","author":{"firstname":"Ron","lastname":"LEsteve"}}']

rdd = sc.parallelize(data)

df = spark.read.json(rdd)

df.printSchema()

Since the data has been displayed in a multiline format, shown in section 1 of Figure 6-13, you can run the following command: display(df.select("booktitle","author.firstname","author.lastname")) to select the fields that you want to display in tabular format, shown in section 2 of Figure 6-13. Alternatively, you can run this command to display the print format results shown in section 3 of Figure 6-13: df.select("booktitle","author.firstname","author.lastname").show()

The last block of code in this section of the script will list the files stored in the databricks/driver directory.

#Code to import the urllib package and read the contents of the specified URL zip file.

import urllib

urllib.request.urlretrieve("https://resources.lendingclub.com/LoanStats3a.csv.zip", "/tmp/LoanStats3a.csv.zip")

#Code the unzip the url zip file

%sh

unzip /tmp/LoanStats3a.csv.zip

tail -n +2 LoanStats3a.csv > temp.csv

rm LoanStats3a.csv

#Code to list the files in the folder

%fs ls file:/databricks/driver/

You can run the following script to move the file from the databricks/driver folder to your mounted ADLSgen2 account. The second section of the code will load the unzipped CSV file into a dataframe and display it. The final code in this section shows an option for running the %sh magic command to unzip a .zip file, when needed.

#Code to move the file from databricks/driver folder to your mounted ADLSgen2 account.

dbutils.fs.mv("file:/databricks/driver/temp.csv", "/mnt/raw/LoanStats3a.csv")

#Code to load the unzipped csv file in your ADLSgen2 account into a dataframe.

df = spark.read.format("csv").option("inferSchema", "true").option("header","true").load("/mnt/raw/LoanStats3a.csv")

display(df)

#Option for unzipping a zip file in your ADLSgen2 account

%sh

unzip mnt/raw/emp.zip

The following code will achieve the same goals as the PySpark script in the XML section, with a few obvious syntactical differences. This shows how both PySpark and Scala can achieve the same outcomes.

// Infer schema

import com.databricks.spark.xml.\_ // Add the DataFrame.read.xml() method

val df = spark.read

.option("rowTag", "book")

.xml("dbfs:/books.xml")

val selectedData = df.select("author", "\_id")

selectedData.write

.option("rootTag", "books")

.option("rowTag", "book")

.xml("dbfs:/newbooks.xml")

The following code shows how to query nested json format data using SQL.

SELECT

book:bookid.author,

book:bookid.title,

book:bookid.genre,

FROM book;

The code below shows an example of how a function could be created to return the max amount of books sold for a specified book id.

/\* function to return max amount of books sold  
\*\* for the specified book\_id  
\*/  
  
CREATE FUNCTION udf\_max\_copies\_sold\_for\_title (@book\_id CHAR(6))  
RETURNS INT  
AS  
BEGIN

DECLARE @qty INT

-- initialize the variable at 0:

SELECT @qty = 0

SELECT

@qty = MAX(qty)

FROM sales

WHERE

book\_id = @book\_id

/\* If there are no books sold for book\_id specified

\*\* then return 0:

\*/

RETURN ISNULL(@qty, 0)

END

For performance associated with sums and counts, know when to use the following functions instead of the standard groupByKey():

1. **groupByKey:** This key may cause ‘out of disk problems’ as data is sent over the network and collected on the reduced workers. The Syntax for a Standard groupByKey may look like this:

sparkContext.textFile("hdfs://")

.flatMap(line => line.split(" ") )

.map(word => (word,1))

.groupByKey()

.map((x,y) => (x,sum(y)))

1. **reduceByKey:** Use when you need to group by a key on a partitioned or distributed dataset to aggregate values for each key. Data are combined at each partition, with only one output for one key at each partition to send over the network. The following code shows an example of reduceByKey:

sparkContext.textFile("hdfs://")

.flatMap(line => line.split(" "))

.map(word => (word,1))

.reduceByKey((x,y)=> (x+y))

1. **combineByKey:** Use when you need to group by a key on a partitioned or distributed dataset to aggregate values for each key with different inputs and outputs for the ‘reduce’ functions. The following code shows an example of combineByKey

val result = rdd.combineByKey(

(v) => (v,1),

( (acc:(Int,Int),v) => acc.\_1 +v , acc.\_2 +1 ) ,

( acc1:(Int,Int),acc2:(Int,Int) => (acc1.\_1+acc2.\_1) , (acc1.\_2+acc2.\_2))

).map( { case (k,v) => (k,v.\_1/v.\_2.toDouble) })

result.collect.foreach(println)

1. **foldByKey:** Use when you need to aggregate values based on keys to perform an operation which yields a zero value. The following code shows an example of foldByKey:

val Scores = Array(("Ron", 90.0), ("Cayden", 100.0), ("Christina", 93.0), ("Ron", 95.0), ("Cayden", 70.0), ("Dee", 98.0))

val scoreData = sc.parallelize(Scores).cache()

scoreData.foldByKey(0)(\_+\_).collect