Ingesting Microsoft Excel using Apache Spark Structured Streaming

Everyone who has been into building data ingestions for the Internet of Things (IoT) using Apache Spark would have been very well versed with Apache Spark structured streaming.

You may have used it to stream data from various event streaming technologies like Kafka or Azure EventHub and use structured streaming to ingest data using continuous processing or time-based trigger. Structured streaming also provides trigger modes that enable batch processing once or whenever new data is available.

Another technology worth mentioning here is Databricks Autoloader, introduced back in 2020, which helps to incrementally ingest a variety of data sources into cloud storage using the same Apache Spark structured streaming functionalities with an added new streaming source called "cloudFiles". This automatically set up file notification services from the input directory and processes new files as they arrive.

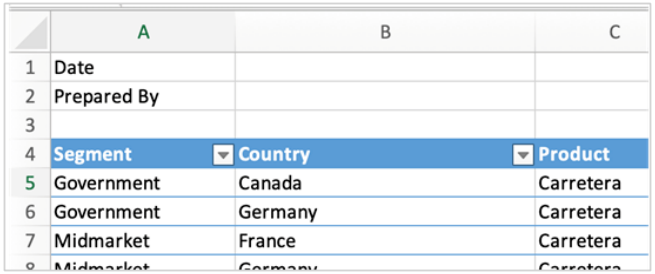
While this short article is not a tutorial on structure streaming and Databricks Autoloader, these features have impacted streaming workloads for both structured and non-structured files and have been used extensively in our Datalake Accelerator.

A very common scenario in an Extract, Transform and Load (ETL) or sometimes described as Extract, Load and Transform (ELT) dataflow, we ingest data from relational databases; we also have scenarios where users drop external or internally created Microsoft Excel (XLSX) files and used as part of data for analytics purpose. While the frequency of dropping these XLSX files can be determined, we would often like to automate it as part of an ETL/ELT orchestration, not manually.

Using Apache Spark structured streaming and Databricks Autoloader (if running in a Databricks environment), we can create a process that automatically picks up the new XLSX file and save it as a delta format table. We then can combine or transform this delta table and use it as a data source for analytical purposes.

Assuming we have some XLSX files that have been dropped in an Azure storage container as below.

This XLSX file we are using is Financial Sample.xlsx which is a fictitious company sales report that looks like this:



	A	B	C
1	Date		
2	Prepared By		
3			
4	Segment	Country	Product
5	Government	Canada	Carretera
6	Government	Germany	Carretera
7	Midmarket	France	Carretera

We can use the following PySpark code to initiate a streaming process from the Azure storage container.

```
landing = (spark
            .readStream
            .option("rescueDataColumn", "_rescued_data")
            .schema(xlsx_files)
            .format("binaryFile")
            .load(sourcePath)
)

raw = (landing
            .writeStream
            .queryName("xlsx_file")
            .trigger(once=True)
            .format("delta")
            .outputMode("append")
            .option("checkpointLocation", checkpointPath)
            .start(deltaPath)
)
```

In the above code snippet, I have defined the input file format as “binaryFile”, and the “sourcePath” points to the user-drop/excel location. I have also used a schema to infer the schema for the structure of the dataframe that is used when we persist the dataframe as a delta table in the “writeStream”. The rest are regular syntax for structured streaming. The above snippet also has some “cloudFiles” options missing, as I intentionally left that out so it can work in any PySpark environment, e.g., Synapse notebooks using Apache Spark pool.

At this point, we are not ingesting the XLSX file per see, but instead, the content of the user-drop/excel folder; thus, the schema will look as follows.

```
xlsx_files = "`path` STRING, `modificationTime` TIMESTAMP, `length` LONG, `content` BINARY"
```

When we run the streaming code above, we will see the result below:

And this is what our delta table would look like. The column of interest will be the content column.

```
1  df = spark.read.format("delta").load(deltaPath)
2  df.show()
```

▸ (6) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [path: string, modificationTime: timestamp ... 2 more fields]

```
+--------------------+-------------------+------+--------------------+
|                path|   modificationTime|length|             content|
+--------------------+-------------------+------+--------------------+
|abfss://user-drop...|2023-01-29 20:30:27|153459|[50 4B 03 04 14 0...|
|abfss://user-drop...|2023-01-29 19:26:34| 61720|[50 4B 03 04 14 0...|
+--------------------+-------------------+------+--------------------+
```

From here, assuming we have some metadata configuration file that describes what XLSX file we are interested in and how the schema (columns of the XLSX file), we can filter the above delta table using the "path" column and take the "content" column convert is as a "byte" object.

```
1. # The xlsx file we want to extract
2. xlsx_file = "Financial Sample"
3.
4. # Filter the row with the path that looks like xlsx_file
5. xlsx_table = df.where(f"path LIKE '%{table}%'")
6.
7. # Collect the content and convert the "content" bytearray to bytes
8. stream = df.collect()[0].content
9. byteObject = bytes(stream)
10.
```

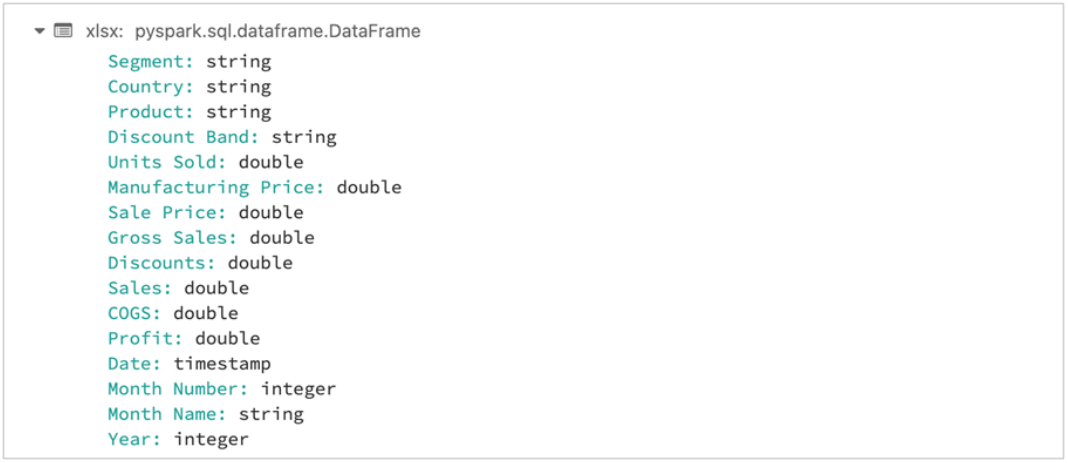
Anyone using PySpark to read the XLSX file will know that spark-excel_2.12-3.3.1_0.18.5.jar from crealytics can be used, and it worked perfectly well in many of our previous projects, but getting it to work in a streaming environment, can be quite challenging. So instead of using the above library, I am using Pandas (part of Apache Spark framework) native read_excel() to read "byteObject", which is the converted "stream" (in bytes) as input to Pandas read_excel().

```
1.
```

```
 excel_data = pd.read_excel(
2.     byteObject,
3.     skiprows=3,         # The actual data starts at Row 4
4.     sheet_name="Sheet1" # The worksheet that we are interested in
5. )
```

From here, we convert the Pandas dataframe to Spark dataframe by creating a new Apache Spark dataframe with a schema that describes the XLSX file.

```
1. xlsx_schema = "`Segment` string, `Country` string, `Product` string, `Discount Band` string,
`Units Sold` double, `Manufacturing Price` double, `Sale Price` double, `Gross Sales` double,
`Discounts` double, `Sales` double, `COGS` double, `Profit` double, `Date` timestamp, `Month
Number` int, `Month Name` string, `Year` int"
2.
3. xlsx = spark.createDataFrame(excel_data, schema=xlsx_schema)
4.
```

I hope this has provided some insights into how we can ingest Microsoft Excel files via Apache Spark structured streaming.

This feature is fundamental to providing a common ingesting strategy for structured and unstructured requirements built into our Lakehouse Accelerator, simplifying data ingestion with data accuracy and rapidly delivering value from data.

If your business needs help with their data quality and accuracy, please contact a Fujitsu Data & AI specialist now.

Contact
Fujitsu Data & AI
+61 3 9924 3000