Powering Data Pipelines: XML Ingestion with Databricks Autoloader

Many of us have and maybe still use XML as a standard format for various applications and industries in web services, APIs, and data integration scenarios for exchanging data between different systems, platforms, and programming languages.

XML is considered a structured data format. It provides a well-defined structure for organising and representing data using tags and hierarchical relationships. XML documents follow a set of rules and guidelines, allowing data to be organised in a logical and consistent manner.

Unlike unstructured data, which lacks a predefined organisation or schema, XML enforces a hierarchical structure and defines the relationships between different elements. This structure enables data validation, querying, and processing based on the defined structure and rules.

While this short article is not a tutorial on structure streaming and Autoloader, these features have impacted streaming workloads for both structured and non-structured files and have been used extensively in our Datalake Accelerator. There are many benefits in using Autoloader but one of the key factors why it is changing ingestion landscape is because AutoLoader incrementally and efficiently processes new data files as they arrive in cloud storage without any additional setup.

A very common scenario in an Extract, Transform and Load (ETL) or sometimes described as Extract, Load and Transform (ELT) dataflow, is where we use Databricks to request data from REST API or maybe use external data exchange from other applications or data exchanges between organisations and drop into distributed storage like Azure Data Lake Storage or AWS S3 bucket and used as part of data for analytics purpose. While the frequency of dropping these XML files can be determined, we would often like to automate it as part of an ETL/ELT orchestration, not manually.

Assuming we have a simple XML file that has been dropped in an Azure storage container which looks like this

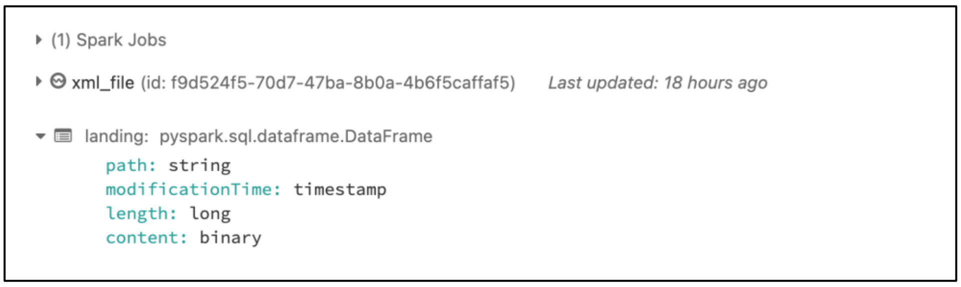
```
<Sales>
          <Orders>
                    <Order id="03413">
                              <CustomerID>9a132a1c-ebf2-4e31-b38d-ce00169c201a</CustomerID>
                              <OrderDate>1607247470559</OrderDate>
                              <Note>advance couple medicines rally pts</Note>
                              <comment>feeds</comment>
                              <OrderDetails>
                                        <OrderDetail>
                                                  <ProductID>MAZOLA</ProductID>
                                                  <UnitPrice>90.61</UnitPrice>
                                                  <Quantity>36</Quantity>
                                        </OrderDetail>
                              </OrderDetails>
                    </Order>
          </Orders>
          <Orders>
</Sales>
```

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

```
landing = (spark
           .readStream
           .format("cloudFiles")
           .option("cloudFiles.format", "binaryFile")
           .load(sourcePath)
)

bronze = (landing
           .writeStream
           .queryName("xml_file")
           .trigger(availableNow=True)
           .format("delta")
           .outputMode("append")
           .foreachBatch(foreachbatch_xml)
           .option("checkpointLocation", checkpointPath)
           .start()
)

```

In the above code snippet, I have defined the Autoloader file type to “cloudFiles”, and from here, we can add options to include how the file is handled during the ingestion. Here I have added an option to treat the input file format as “binaryFile”, and the “sourcePath” points to the location of the source.

Using the above “readStream”, my “landing” data frame would look like this when we run the Autoloader code above.



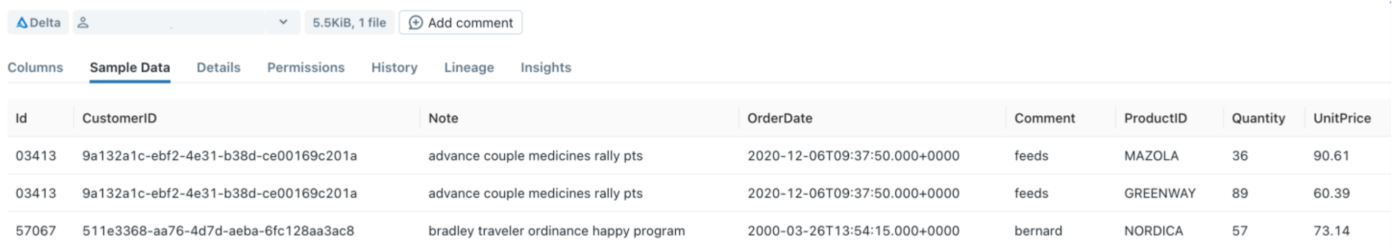
```
▸ (1) Spark Jobs
▸ xml_file (id: f9d524f5-70d7-47ba-8b0a-4b6f5caffaf5)     Last updated: 18 hours ago
▾ landing: pyspark.sql.dataframe.DataFrame
      path: string
      modificationTime: timestamp
      length: long
      content: binary
```

You may also notice that for “writeStream”, I have included a “foreachbatch” option that would allow me to specify a function that is executed on the output data of every micro-batch of the streaming query. It takes two parameters: a data frame or Dataset that has the output data of a micro-batch and the unique ID of the micro-batch. This is necessary here; as you recall, I have ingested the XML file via “readStream” as a binary file, and it is in this function that I will convert the “content” column to string and use the “xmlToDict” library to convert to the XML to a JSON data frame, so that I can format the XML contents to its proper columns.

The code snippet below describes the process and additionally cleaning column names to a proper camel case format before persisting to a delta table using Unity Catalog 3-level names base structure.

```
def foreachbatch_xml(batch_df, batch_id):

    batches = batch_df.collect()
    for xml in batches:
        stream = xml.content
        content = bytes(stream).decode("utf-8")
        df  = spark.read.json(sc.parallelize([json.dumps(xmltodict.parse(content))]))

        bronze = (
            df
                .selectExpr("inline(Sales.Orders.Order)")
                .select("*", explode("OrderDetails.OrderDetail").alias("to_be_drop"))
                .select("*", "to_be_drop.*")
                .withColumn("OrderDate", from_unixtime(col("OrderDate")/1000))
                .drop("OrderDetails", "to_be_drop")

        )
        columns = [camelCase(x) for x in bronze.columns] # sanitize column names to camel case
        bronze = (
            bronze
                .toDF(*columns)
                .selectExpr(
                    "cast(Id as string) Id",
                    "cast(CustomerID as string) CustomerID",
                    "cast(Note as string) Note",
                    "cast(OrderDate as timestamp) OrderDate",
                    "cast(Comment as string) Comment",
                    "cast(ProductID as string) ProductID",
                    "cast(Quantity as int) Quantity",
                    "cast(UnitPrice as decimal(10,2)) UnitPrice"
                )
        )

        bronze.write.mode("overwrite").saveAsTable(name = "samples.sales.order")


```

Sample output from the table in Data Explorer



Id	CustomerID	Note	OrderDate	Comment	ProductID	Quantity	UnitPrice
03413	9a132a1c-ebf2-4e31-b38d-ce00169c201a	advance couple medicines rally pts	2020-12-06T09:37:50.000+0000	feeds	MAZOLA	36	90.61
03413	9a132a1c-ebf2-4e31-b38d-ce00169c201a	advance couple medicines rally pts	2020-12-06T09:37:50.000+0000	feeds	GREENWAY	89	60.39
57067	511e3368-aa76-4d7d-aeba-6fc128aa3ac8	bradley traveler ordinance happy program	2000-03-26T13:54:15.000+0000	bernard	NORDICA	57	73.14

While there are other ways to parse XML files in the Apache Spark platform, e.g., using Spark-XML driver but I find using “xmlToDict” allows more flexibility and control of how the XML data can be easily managed.

I hope this has provided some insights into how we can ingest XML files via Databricks Autoloader. 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 data quality and accuracy, please contact one of our specialists by emailing us or call 03 9924 3000.

Contact

Fujitsu Data & AI

+61 3 9924 3000