XML Format: User Documentation

This feature is in Private Preview.

Extensible Markup Language (XML) is a markup language for formatting, storing and sharing data in textual format. It defines a set of rules for serializing data ranging from documents to arbitrary data structures. So far Databricks' users had to load an external package 'spark-xml' to read and write XML data. But it didn't work well with streaming (autoloader, for example) and serverless and lacked advanced capabilities like schema evolution that are available with other text formats like csv and json.

The new native XML File Format support in Databricks enables ingestion, querying and parsing of XML data for batch processing or streaming. It can automatically infer and evolve schema and data types, support SQL expressions like `from_xml`, as well as generate XML documents. It doesn't require any external jars and works seamlessly with Auto Loader, `read_files`, `COPY INTO` and DLT.

This document describes how to read and write XML data with Databricks.

Loading XML Data into Databricks

Databricks offers a variety of ways to help you load XML data into a lakehouse backed by Delta Lake. These include Auto Loader and COPY INTO. XML support will be added to Add Data UI shortly. In addition, the following SQL built-in functions are also available to read XML data:

- read_files: Reads files under a provided location and returns the data in tabular form.
- from_xml(xmlStr, schema[, options]): Returns a struct value with the xmlStr and schema.
- schema_of_xml(xmlStr[, options]): Returns the schema of a XML string in DDL format.
- to_xml(expr[, options]): Returns a XML string with the `struct` specified in `expr`. (Not available in private preview)

Prerequisites

Native XML is available with Databricks Runtime 14.1 and above. During the private preview, users have to explicitly enable XML using the conf shown below:

```
spark.conf.set("spark.databricks.sql.nativeXmlDataSourcePreview.enabled", true)
```

The above conf can be set in the Notebook or in the Spark config at the time of cluster creation.

Parsing XML Record

XML specification mandates a well-formed structure. However, this specification doesn't immediately map to a tabular format. You have to specify the name of an XML element via the rowTag option to indicate the beginning and end of a DataFrame Row record. The rowTag element becomes the top level 'Struct'. The child elements of rowTag become the fields of the top-level Struct. You can specify the schema for this record or let it be inferred automatically. Since the parser only examines the rowTag elements, DTD, external and internal entities are disabled by default.

Here is a sample XML data and its parsed results with different rowTag option:
RowTag as books (Top level element is an array type)

```
val xmlString = """
  <books>
    <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
    </book>
    <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
    </book>
  </books>
"""
val xmlPath = inputPath + "books.xml"
dbutils.fs.put(xmlPath, xmlString)
val df = spark.read.option("rowTag", "book").xml(xmlPath)
df.printSchema()
df.show(truncate=false)
root
 |-- book: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- _id: string (nullable = true)
 |    |    |-- author: string (nullable = true)
 |    |    |-- title: string (nullable = true)

+--------------------------------------------------------------------------------+
|book                                                                            |
+--------------------------------------------------------------------------------+
|[{bk103, Corets, Eva, Maeve Ascendant}, {bk104, Corets, Eva, Oberon's Legacy}]|
+--------------------------------------------------------------------------------+
```

RowTag as book (Top level element is a struct type)

```
val df = spark.read.option("rowTag", "book").xml(xmlPath)

root
 |-- _id: string (nullable = true)
 |-- author: string (nullable = true)
 |-- title: string (nullable = true)

+-----+-----------+---------------+
|_id  |author     |title          |
+-----+-----------+---------------+
|bk103|Corets, Eva|Maeve Ascendant|
|bk104|Corets, Eva|Oberon's Legacy|
+-----+-----------+---------------+
```

Data Source Option

Data source options of XML can be set via:

- the .option/.options methods of
  - DataFrameReader
  - DataFrameWriter
  - DataStreamReader
  - DataStreamWriter
- the built-in functions below
  - from_xml
  - to_xml (not available in private preview)
  - schema_of_xml
- OPTIONS clause at [CREATE TABLE USING DATA_SOURCE]

Option	Description	Scope
rowTag	The row tag of your xml files to treat as a row. For example, in this xml: `<books> <book><book> ...</books>` the appropriate value would be book. Default: ROW	read
samplingRatio	Defines fraction of rows used for schema inferring. XML built-in functions ignore this option. Default is 1.0.	read
excludeAttribute	Whether to exclude attributes in elements. Default: false	read

mode	Allows a mode for dealing with corrupt records during parsing. PERMISSIVE: when it meets a corrupted record, puts the malformed string into a field configured by columnNameOfCorruptRecord, and sets malformed fields to null. To keep corrupt records, an user can set a string type field named columnNameOfCorruptRecord in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. When inferring a schema, it implicitly adds a columnNameOfCorruptRecord field in an output schema. DROPMALFORMED: ignores the whole corrupted records. This mode is unsupported in the XML built-in functions. FAILFAST: throws an exception when it meets corrupted records.	read
inferSchema	If true, attempts to infer an appropriate type for each resulting DataFrame column. If false, all resulting columns are of string type. Default is true. XML built-in functions ignore this option.	read
columnNameOfCorruptRecord	Allows renaming the new field having a malformed string created by PERMISSIVE mode. Default: spark.sql.columnNameOfCorruptRecord	read
attributePrefix	The prefix for attributes to differentiate attributes from elements. This will be the prefix for field names. Default is _. Can be empty for reading XML, but not for writing.	read / write
valueTag	The tag used for the value when there are attributes in the element having no child. Default is _VALUE.	read / write
encoding	For reading, decodes the XML files by the given encoding type. For writing, specifies encoding (charset) of saved XML files. XML built-in functions ignore this option. Default is UTF-8	read / write
ignoreSurroundingSpaces	Defines whether surrounding whitespaces from values being read should be skipped. Default is false.	read
rowValidationXSDPath	Path to an optional XSD file that is used to validate the XML for each row individually. Rows that fail to validate are treated like parse errors as above. The XSD does not otherwise affect the schema provided, or inferred.	read
ignoreNamespace	If true, namespaces prefixes on XML elements and attributes are ignored. Tags <abc:author> and <def:author> would, for example, be treated as if both are just <author>. Note that, at the moment, namespaces cannot be ignored on the rowTag element, only its	read

	children. Note that XML parsing is in general not namespace-aware even if false. Defaults to false.	
timestampFormat	Custom timestamp format string that follows the datetime pattern format. This applies to timestamp type. Default: yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]	read / write
dateFormat	Custom date format string that follows the datetime pattern format. This applies to date type. Default: yyyy-MM-dd	read / write
locale	Sets a locale as a language tag in IETF BCP 47 format. For instance, locale is used while parsing dates and timestamps. Default: en-US	read
rootTag	Root tag of the xml files. For example, in <books> <book><book> ...</books>, the appropriate value would be books. It can include basic attributes by specifying a value like books foo="bar". Default is ROWS.	write
declaration	Content of XML declaration to write at the start of every output XML file, before the rootTag. For example, a value of foo causes <?xml foo?> to be written. Set to empty string to suppress. Defaults to version="1.0" encoding="UTF-8" standalone="yes".	write
arrayElementName	Name of XML element that encloses each element of an array-valued column when writing. Default is item	write
nullValue	Sets the string representation of a null value. Default is string null. When this is null, it does not write attributes and elements for fields.	read/ write
compression	Compression codec to use when saving to file. This can be one of the known case-insensitive shortened names (none, bzip2, gzip, lz4, snappy and deflate). XML built-in functions ignore this option. Default: none	write

Other generic options can be found in Generic File Source Options and Auto Loader options.

XSD Support

Users can optionally validate each individual XML record returned by the 'record tokenizer' by a XSD (XML Schema Definition). The XSD file is specified by `rowValidationXSDPath` option. The XSD does not otherwise affect the schema provided, or inferred. A record that fails the validation will be marked as "corrupted" and handled based on the corrupt record handling mode option described in the option section.

The utility XSDToSchema can be used to extract a Spark DataFrame schema from some XSD files. It supports only simple, complex and sequence types, and only basic XSD functionality.

```
import org.apache.spark.sql.catalyst.xml.XSDToSchema
import java.nio.file.Paths

val xsdPath = inputPath + "books.xsd"
val xsdString = """<?xml version="1.0" encoding="UTF-8" ?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="book">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="author" type="xs:string" />
          <xs:element name="title" type="xs:string" />
          <xs:element name="genre" type="xs:string" />
          <xs:element name="price" type="xs:decimal" />
          <xs:element name="publish_date" type="xs:date" />
          <xs:element name="description" type="xs:string" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:string" use="required" />
      </xs:complexType>
    </xs:element>
  </xs:schema>"""

dbutils.fs.rm(xsdPath, true)
dbutils.fs.put(xsdPath, xsdString)

val schema1 = XSDToSchema.read(xsdString)
val schema2 = XSDToSchema.read(Paths.get("/dbfs" + xsdPath))
```

The following table shows the conversion of XSD data types to Spark data types:

XSD Data Types	Spark Data Types
boolean	BooleanType
decimal	DecimalType
unsignedLong	DecimalType(38, 0)
double	DoubleType
float	FloatType

byte	ByteType
short, unsignedByte	ShortType
integer, negativeInteger, nonNegativeInteger, nonPositiveInteger, positiveInteger, unsignedShort	IntegerType
long, unsignedInt	LongType
date	DateType
dateTime	TimestampType
Others	StringType

Parsing Nested XML

XML data in a string-valued column in an existing DataFrame can be parsed with schema_of_xml and from_xml that returns the schema and the parsed results as a new struct column respectively.

XML data passed as argument to schema_of_xml and from_xml must be a single well-formed XML record.

Syntax:
```
schema_of_xml(xmlStr [, options] )
```

Arguments
- xmlStr: A STRING expression specifying a single well-formed XML record.
- options: An optional MAP<STRING,STRING> literal specifying directives.

Returns
A STRING holding a definition of a struct with n fields of strings where the column names are derived from the XML element and attribute names. The field values hold the derived formatted SQL types.

Syntax:
```
from_xml(xmlStr, schema [, options])
```

Arguments
- xmlStr: A STRING expression specifying a single well-formed XML record.
- schema: A STRING expression or invocation of schema_of_xml function.
- options: An optional MAP<STRING,STRING> literal specifying directives.

Returns
A struct with field names and types matching the schema definition.

Schema must be defined as comma-separated column name and data type pairs as used in for example CREATE TABLE. Most options shown in the Data Source options are applicable with the following exception:

- rowTag: As there is only one XML record, the rowTag option is not applicable.
- mode (default: PERMISSIVE): allows a mode for dealing with corrupt records during parsing.
  - PERMISSIVE: when it meets a corrupted record, puts the malformed string into a field configured by columnNameOfCorruptRecord, and sets malformed fields to null. To keep corrupt records, you can set a string type field named columnNameOfCorruptRecord in an user-defined schema. If a schema does not have the field, it drops corrupt records during parsing. When inferring a schema, it implicitly adds a columnNameOfCorruptRecord field in an output schema.
  - FAILFAST: throws an exception when it meets corrupted records.

Structure Conversion

Due to the structure differences between `DataFrame` and XML, there are some conversion rules from XML data to `DataFrame` and from `DataFrame` to XML data. Note that handling attributes can be disabled with the option `excludeAttribute`.

Conversion from XML to DataFrame

- **Attributes**: Attributes are converted as fields with the heading prefix, `attributePrefix`.

```
<one myOneAttrib="AAAA">
    <two>two</two>
    <three>three</three>
</one>
```

  produces a schema below:

```
root
 |-- _myOneAttrib: string (nullable = true)
 |-- two: string (nullable = true)
 |-- three: string (nullable = true)
```

- **Value in an element that has no child elements but attributes:** The value is put in a separate field, `valueTag`.

```
<one>
    <two myTwoAttrib="BBBBB">two</two>
    <three>three</three>
</one>
```

  produces a schema below:

```
root
 |-- two: struct (nullable = true)
 |    |-- _VALUE: string (nullable = true)
 |    |-- _myTwoAttrib: string (nullable = true)
 |-- three: string (nullable = true)
```

Conversion from DataFrame to XML

- **Element as an array in an array:** Writing a XML file from `DataFrame` having a field `ArrayType` with its element as `ArrayType` would have an additional nested field for the element. This would not happen in reading and writing XML data but writing a `DataFrame` read from other sources. Therefore, roundtrip in reading and writing XML files has the same structure but writing a `DataFrame` read from other sources is possible to have a different structure.
  `DataFrame` with a schema below:

```
 |-- a: array (nullable = true)
 |    |-- element: array (containsNull = true)
 |    |    |-- element: string (containsNull = true)
```

  with data below:

```
+------------------------------------+
|                                   a|
+------------------------------------+
|[WrappedArray(aa), WrappedArray(bb)]|
+------------------------------------+
```

  produces a XML file below:

```
<a>
    <item>aa</item>
</a>
<a>
    <item>bb</item>
</a>
```

  The element name of the unnamed array in the DataFrame is specified by the option arrayElementName (Default is item.)

Rescued data column

The rescued data column ensures that you never lose or miss out on data during ETL. You can enable the rescued data column to capture any data that wasn't parsed because one or more fields in a record have one of the following issues:

- Absent from the provided schema.
- Does not match the data type of the provided schema.
- Has a case mismatch with the field names in the provided schema.

The rescued data column is returned as a JSON document containing the columns that were rescued, and the source file path of the record. To remove the source file path from the rescued data column, you can set the SQL configuration:
spark.conf.set("spark.databricks.sql.rescuedDataColumn.filePath.enabled", "false").
You can enable the rescued data column by setting the option rescuedDataColumn to a column name when reading data, such as _rescued_data with spark.read.option("rescuedDataColumn", "_rescued_data").format("xml").load(<path>).

The XML parser supports three modes when parsing records: PERMISSIVE, DROPMALFORMED, and FAILFAST. When used together with rescuedDataColumn, data type mismatches do not cause records to be dropped in DROPMALFORMED mode or throw an error in FAILFAST mode. Only corrupt records - that is, incomplete or malformed XML - are dropped or throw errors.

Schema inference and evolution in Auto Loader

For detailed discussion on this topic and applicable options, please refer to this article.
You can configure Auto Loader to automatically detect the schema of loaded XML data, allowing you to initialize tables without explicitly declaring the data schema and evolve the table schema as new columns are introduced. This eliminates the need to manually track and apply schema changes over time.
By default, Auto Loader schema inference seeks to avoid schema evolution issues due to type mismatches. For formats that don't encode data types (JSON, CSV and XML), Auto Loader infers all columns as strings (including nested fields in XML files). The Apache Spark DataFrameReader uses different behavior for schema inference, selecting data types for columns in XML sources based on sample data. To enable this behavior with Auto Loader, set the option cloudFiles.inferColumnTypes to true.
Auto Loader detects the addition of new columns as it processes your data. When Auto Loader detects a new column, the stream stops with an UnknownFieldException. Before your stream throws this error, Auto Loader performs schema inference on the latest micro-batch of data and updates the schema location with the latest schema by merging new columns to the end of the schema. The data types of existing columns remain unchanged. Auto Loader supports different modes for schema evolution, which you set in the option cloudFiles.schemaEvolutionMode.

You can use schema hints to enforce the schema information that you know and expect on an inferred schema. When you know that a column is of a specific data type, or if you want to choose a more general data type (for example, a double instead of an integer), you can provide an arbitrary number of hints for column data types as a string using SQL schema specification syntax.

When rescued data column is enabled, fields named in a case other than that of the schema are loaded to the _rescued_data column. Change this behavior by setting the option readerCaseSensitive to false, in which case Auto Loader reads data in a case-insensitive way.
Support for readerCaseSensitive option in XML private preview will be made available shortly.

Examples

These examples use a XML file available for download here:
https://github.com/apache/spark/blob/master/sql/core/src/test/resources/test-data/xml-resources/books.xml

SQL API

XML data source can infer data types:

```
%sql
DROP TABLE IF EXISTS books;
CREATE TABLE books
USING XML
OPTIONS (path "books.xml", rowTag "book");
SELECT * FROM books;
```

You can also specify column names and types in DDL. In this case, schema is not inferred automatically.

```
%sql
DROP TABLE IF EXISTS books;
CREATE TABLE books (author string, description string, genre string, _id string,
price double, publish_date string, title string)
USING XML
OPTIONS (path "books.xml", rowTag "book");
```

Load XML using COPY INTO

```
%sql
DROP TABLE IF EXISTS books;
CREATE TABLE IF NOT EXISTS books;

COPY INTO books
FROM "/FileStore/xmltestDir/input/books.xml"
FILEFORMAT = XML
FORMAT_OPTIONS ('mergeSchema' = 'true', 'rowTag' = 'book')
COPY_OPTIONS ('mergeSchema' = 'true');
```

Scala API

```
val df = spark.read
  .option("rowTag", "book")
  .xml("books.xml")

val selectedData = df.select("author", "_id")
selectedData.write
  .option("rootTag", "books")
  .option("rowTag", "book")
  .xml("newbooks.xml")
```

You can manually specify the schema when reading data:

```
import org.apache.spark.sql.types.{StructType, StructField, StringType, DoubleType}

val customSchema = StructType(Array(
  StructField("_id", StringType, nullable = true),
  StructField("author", StringType, nullable = true),
  StructField("description", StringType, nullable = true),
  StructField("genre", StringType, nullable = true),
  StructField("price", DoubleType, nullable = true),
  StructField("publish_date", StringType, nullable = true),
  StructField("title", StringType, nullable = true)))


val df = spark.read
  .option("rowTag", "book")
  .schema(customSchema)
  .xml("books.xml")

val selectedData = df.select("author", "_id")
selectedData.write
  .option("rootTag", "books")
  .option("rowTag", "book")
  .xml("newbooks.xml")
```

Python API

```
%python
df = spark.read.format('xml').options(rowTag='book').load('books.xml')
df.select("author", "_id").write \
  .format('xml') \
  .options(rowTag='book', rootTag='books') \
  .save('newbooks.xml')
```

You can manually specify the schema when reading data:

```
%python
from pyspark.sql.types import *

customSchema = StructType([
  StructField("_id", StringType(), True),
  StructField("author", StringType(), True),
  StructField("description", StringType(), True),
  StructField("genre", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("publish_date", StringType(), True),
  StructField("title", StringType(), True)])

df = spark.read \
  .format('xml') \
  .options(rowTag='book') \
  .load('books.xml', schema = customSchema)

df.select("author", "_id").write \
  .format('xml') \
  .options(rowTag='book', rootTag='books') \
  .save('newbooks.xml')
```

R API

```
%r
df <- loadDF("books.xml", source = "xml", rowTag = "book")
# In this case, `rootTag` is set to "ROWS" and `rowTag` is set to "ROW".
saveDF(df, "newbooks.xml", "xml", "overwrite")
%r
customSchema <- structType(
  structField("_id", "string"),
  structField("author", "string"),
  structField("description", "string"),
  structField("genre", "string"),
  structField("price", "double"),
  structField("publish_date", "string"),
  structField("title", "string"))

df <- loadDF("books.xml", source = "xml", schema = customSchema, rowTag = "book")
# In this case, `rootTag` is set to "ROWS" and `rowTag` is set to "ROW".
saveDF(df, "newbooks.xml", "xml", "overwrite")
```

Read XML with Row Validation

```
val df = spark.read
  .option("rowTag", "book")
  .option("rowValidationXSDPath", xsdPath)
  .xml(inputPath)
df.printSchema
```

Parsing Nested XML (from_xml and schema_of_xml)

```
import org.apache.spark.sql.functions.{from_xml,schema_of_xml,lit}

val xmlData = """
  <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>
  </book>""".stripMargin
val df = Seq((8, xmlData)).toDF("number", "payload")
val schema = schema_of_xml(xmlData)
val parsed = df.withColumn("parsed", from_xml($"payload", schema))
parsed.printSchema()
parsed.show()
```

From_xml and schema_of_xml with SQL API

```
%sql
SELECT from_xml('
  <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>
  </book>',
  schema_of_xml('
  <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>
  </book>')
);
```

Loading XML with Auto Loader

```
val query = spark
  .readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "xml")
  .option("rowTag", "book")
  .option("cloudFiles.inferColumnTypes", true)
  .option("cloudFiles.schemaLocation", schemaPath)
  .option("cloudFiles.schemaEvolutionMode", "rescue")
  .load(inputPath)
  .writeStream
  .format("delta")
  .option("mergeSchema", "true")
  .option("checkpointLocation", checkPointPath)
  .trigger(Trigger.AvailableNow())
query.start(outputPath).awaitTermination()
val df = spark.read.format("delta").load(outputPath)
df.show()
```