Efficient Processing of XML Documents in Hadoop Map Reduce

Dmitry Vasilenko, Mahesh Kurapati
Business Analytics
IBM
Chicago, USA
dvasilen@us.ibm.com, mkurapati@us.ibm.com

***Abstract*— XML has dominated the enterprise landscape for fifteen years and still remains the most commonly used data format. Despite its popularity the usage of XML for "Big Data" is challenging due to its semi-structured nature as well as rather demanding memory requirements and lack of support for some complex data structures such as maps. While a number of tools and technologies for processing XML are readily available the common approach for map-reduce environments is to create a "custom solution" that is based, for example, on Apache Hive User Defined Functions (UDF). As XML processing is the common use case, this paper describes a generic approach to handling XML based on Apache Hive architecture. The described functionality complements the existing family of Hive serializers/deserializers for other popular data formats, such as JSON, and makes it much easier for users to deal with the large amount of data in XML format.**

Keywords- XML, Apache Hadoop, Apache Hive, Map-Reduce, VTD-XML, XPath.

I. INTRODUCTION

While the enthusiasm around XML as the universal data format readable by humans and computers appears to have subsided [1], XML still remains the widely accepted data format of choice. As a result, there is a growing demand for an efficient processing of large amount of data stored in XML using Apache Hadoop map reduce functionality. The most common approach to process XML data is to introduce a "custom" solution based on the user defined functions or scripts. The typical choices vary from introducing an ETL process for extracting the data of interest to transformations of XML into other formats that are natively supported by Hive. Another possibility is to utilize Apache Hive XPath UDFs but these functions can only be used in Hive views and SELECT statements but not in the table CREATE DDL. The comprehensive survey of other available technologies for processing XML can be found in [2]. The approach proposed in this document enables the user to define XML extraction rules directly in the Hive table CREATE statement by introducing a new XML serializer/deserializer (*SerDe)*. Additionally, a specialization of the Apache Hadoop *TextInputFormat* is introduced to enable parallel processing of plain or compressed XML files.

II. ARCHITECTURAL OVERVIEW

To support efficient processing of the large amount of XML data the proposed implementation relies on the Apache Hadoop Map Reduce stack. The Apache Hive data warehouse allows the user to create a custom *SerDe* [3] to handle a particular type of data. Additionally, the user can implement Hadoop *InputFormat* to create appropriate input splits and associated record readers.

A. Collaboration Diagram

The collaboration diagram for the proposed implementation is shown below.

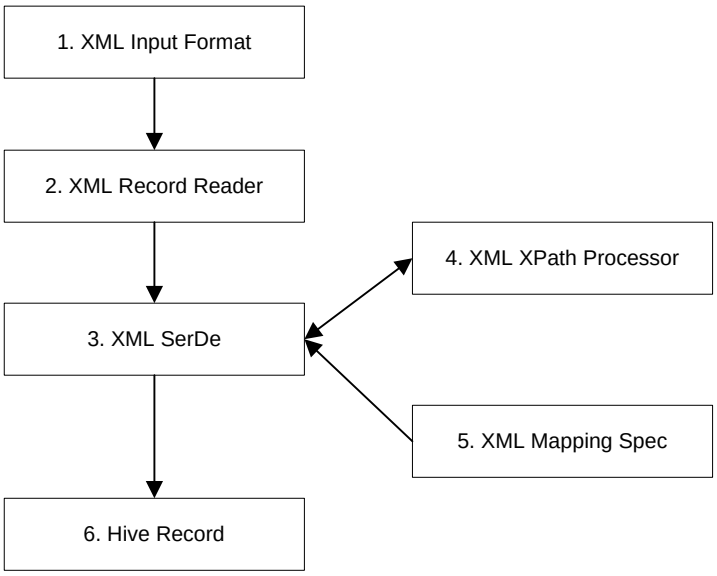

Figure 1. XML SerDe collaboration diagram.

1) XML Input Format. The input format implementation splits up the input files to the XML fragments defined by the start and stop byte sequence for the given XML tag. The implementation evolved from the open source *XmlInputFormat* from Apache Mahout project [4] to the collection of input format classes to deal with compressed and uncompressed XML data. Splittable compressed formats such as BZ2 and CMX can be efficiently processed by a number of Hadoop mappers in parallel. The input splits generated by the XML *InputFormat* are used by the XML reader to create text records for further processing.

2) XML Record Reader. The record reader iterates over the text records based on the start and stop sequence and is used by the XML *SerDe* to transform the records to Hive rows.

3) XML *SerDe*. The XML deserializer interacts with the XML XPath Processor to query XML and generate values for the Hive rows. The transformation specification for the processor is stored in Hive *SERDEPROPERTIES* as described below.

4) XML XPath Processor. The XML processor is a pluggable component implementing *XmlProcessor* interface [5]. The implementation transforms the XML text to the data types supported by the Apache Hive. At the time of this writing there are two publicly available implementations of this interface: one is based on the JDK XPath [5] and another is based on the "Virtual Token Descriptor" XML (VTD-XML) [6].

5) XML Mapping Spec. The specification defines information necessary for mapping of XPath query results to Hive column values as well as a set of conversion rules from XML elements to the Hive map data types.

6) Hive Record. The Hive record for the given XML input.

B. Hive Table Specification

The Hive CREATE TABLE statement for the proposed XML *SerDe* is defined as follows.

```
CREATE [EXTERNAL] TABLE <table_name> (<column_specifications>)
ROW FORMAT SERDE "com.ibm.spss.hive.serde2.xml.XmlSerDe"
WITH SERDEPROPERTIES (
    ["xml.processor.class"="<xml_processor_class_name>",]
    "column.xpath.<column_name>"="<xpath_query>",
     ...
    ["xml.map.specification.<element_name>"="<map_specification>"
...
    ]
)
STORED AS
    INPUTFORMAT "<hadoop_input_format>"
    OUTPUTFORMAT "org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat"
[LOCATION "<data_location>"]
TBLPROPERTIES (
    "xmlinput.start"="<start_tag>",
    "xmlinput.end"="<end_tag>"
);
```

The table parameters described below:

1) xml.processor.class defines the java class name of the XPath processor. Defaults to the JDK XPath implementation. There is also publicly available implementation based on VTD-XML.

2) column.xpath.<column_name> defines the XPath query to retrieve the value for the column.

3) xml.map.specification.<element_name> defines the mapping specification for the XML element. The details are discussed in the XML to Hive Mapping section of the document.

4) <hadoop_input_format> defines the implementation class to split-up the input XML files into logical *InputSplits*, each of which is then assigned to an individual mapper.

5) xmlinput.start and *xmlinput.end* define the byte sequence for the start and end of the XML fragment to be processed.

III. XML TO HIVE RECORD MAPPING

As was noted earlier, there are a number of tools and techniques to map the XML markup to the object instances. Support for complex data types in Apache Hive, however, allows the implementor to do such mapping directly, bypassing the intermediate data types generated by the transformation tools. The following sections shows how the proposed implementation deduces the complex types from the results of the XPath queries.

A. Primitive Types

The attribute values or text content of the XML elements can be directly mapped to the Hive primitive types as shown below.

TABLE I. PRIMITIVE TYPE MAPPING

XML	XPath Expression	Hive DDL	Result
<result>03.06.2009</result>	/result/text()	string result	03.06.2009

B. Structures

The XML element can be directly mapped to the Hive structure type so that all the attributes become the data members. The content of the element becomes an additional member of primitive or complex type.

TABLE II. STRUCTURE MAPPING

XML	XPath Expression	Hive DDL	Result
<result name = "DATE">03.06.2009</result>	/result	struct<name:string,result:string>	name: DATE result: 03.06.2009

C. Arrays

The XML sequences of elements can be represented as Hive arrays of primitive or complex type.

TABLE III. ARRAY MAPPING

XML	XPath Expression	Hive DDL	Result
<result>03.06.2009</result> <result>03.06.2010</result> <result>03.06.2011</result>	/result/text()	array<string >	03.06.2009, 03.06.2010, 03.06.2011

D. Modeling Maps in XML

The XML schema does not provide native support for maps. This issue is well understood and there is a proposal to add map type in the W3C XSLT 3.0 Working Draft [7]. There are three common approaches to modeling maps in XML described in the following sections.

1) Element Name to Element Content

The name of the element is used as a key and its content as a value. This is one of the common techniques and is used by default when mapping XML to Hive map types. The obvious limitations with this approach is that the map key can be only of type string and the key names are fixed.

TABLE IV. ELEMENT NAME TO ELEMENT CONTENT MAPPING

XML	Mapping
<entry1>value1</entry1> <entry2>value2</entry2> <entry3>value3</entry3>	entry1->value1 entry2->value2 entry3->value3

2) Attribute Name to Element Content

Another popular approach is to use an attribute value as a key and the element content as a value.

TABLE V. ATTRIBUTE NAME TO ELEMENT CONTENT MAPPING

XML	Mapping
<entry name=”key1”>value1</entry>	key1->value1
<entry name=”key2”>value2</entry>	key2->value2
<entry name=”key3”>value3</entry>	key3->value3

3) Attribute Name to Attribute Value

Yet another approach is to use two XML attributes to model map entries as shown in the following table.

TABLE VI. ATTRIBUTE NAME TO ATTRIBUTE VALUE MAPPING

XML	Mapping
<entry name=”key1” value=”value1”/>	key1->value1
<entry name=”key2” value=”value2”/>	key2->value2
<entry name=”key3” value=”value3”/>	key3->value3

E. Handling Maps

To accommodate the different ways of modeling maps in XML the proposed implementation introduces the following syntax:

"xml.map.specification.<element_name>"="<key>-><value>"

where:

- *element_name* - the name of the XML element to be considered as a map entry
- *key* – the XML node map entry key
- *value* – the XML node map entry value

The map specification for the given XML element should be defined in the *SERDEPROPERTIES* section in the Hive table creation DDL. The keys and values can be specified using the syntax shown in the following table.

TABLE VII. XML MAPPING SYNTAX

Syntax	Example	Description
@attribute	@name	The @attribute specification allows the user to use the value of the attribute as a key or value of the map.
Element	entry	The XML element name can be used as a key or value.
#content	#content	The content of the XML element can be used as a key or value. As the map keys can only be of primitive type the encountered complex content will be converted to string.

IV. XML COMPRESSION

Using compression for the XML data can significantly reduce storage requirements and still allow for optimal performance during processing. To handle the compressed and raw XML data the framework provides three different implementations for the Hadoop *InputFormats* documented below.

TABLE VIII. XML COMPRESSION FORMATS

Input Format Class	Compression
com.ibm.spss.hive.serde2.xml.XmlInputFormat	Uncompressed XML or non-splittable compression formats such as gzip.
com.ibm.spss.hive.serde2.xml.SplittableXmlInputFormat	Splittable bzip2 compression format.
com.ibm.spss.hive.serde2.xml.CmxXmlInputFormat	Splittable IBM InfoSphere BigInsights CMX compression format.

The *com.ibm.spss.hive.serde2.xml.XmlInputFormat* is publicly available [5]. The splittable input formats for BZ2 and IBM CMX compression are distributed with the IBM SPSS Analytic Server [8].

V. CONCLUSION

This paper describes the proposed and implemented approach to handling large amount of data in XML format using map-reduce functionality. The implementation creates logical splits for the input files each of which is assigned to an individual mapper. The mapper relies on the implemented Apache Hive XML *SerDe* to break the split into XML fragments using specified start/end byte sequences. Each fragment corresponds to a single Hive record. The fragments are then handled by the XML processor to extract values for the record columns utilizing specified XPath queries. The reduce phase is not required.

In the course of this work two variants of the XML processors have been implemented and made publicly available. The JDK XPath based implementation can be used for relatively small XML documents. Large XML documents can be processed with VTD-XML based XPath processor [6]. Additionally, implemented Hadoop *InputFormats* allow for processing of plain and compressed XML files.

ACKNOWLEDGMENTS

Authors thank Prof. Arti Arya of PES University, Bangalore, India, for invaluable comments during preparation of this paper.

REFERENCES

[1] Jim Fuller, Big Data and Modern XML, Keynote, XML Amsterdam, 2013.

[2] C. Subhashini and Arti Arya, "A Framework For Extracting Information From Web Using VTD-XML's XPath", International Journal on Computer Science and Engineering (IJCSE), Vol. 4, No. 03, 2012, p. 463-468.

[3] Apache Hive SerDe, https://cwiki.apache.org/confluence/display/Hive/SerDe

[4] Apache Mahout Project, https://mahout.apache.org/

[5] Apache Hive XML SerDe, https://github.com/dvasilen/Hive-XML-SerDe

[6] Apache Hive VTD XML SerDe, https://github.com/dvasilen/Hive-XML-SerDe-VTD

[7] XSL Transformations (XSLT) Version 3.0, W3C Working Draft 12, December 2013.

[8] IBM SPSS Analytic Server, http://www-03.ibm.com/software/products/en/spss-analytic-server

AUTHORS PROFILE

Dmitry Vasilenko, IBM, 200 West Madison Street, Chicago, IL 60606 (dvasilen@us.ibm.com).

Mr. Vasilenko is a Senior Software Engineer in the Business Analytics Department of the IBM Software Group.

He received a M.S. degree in Electrical Engineering from Novosibirsk State Technical University, Russian Federation, in 1986. Before joining IBM SPSS in 1997 Mr. Vasilenko led Computer Aided Design projects in the area of Electrical Engineering at the Institute of Electric Power System and Electric Transmission Networks. During his tenure with IBM Mr. Vasilenko received three technical excellence awards for his work in Business Analytics. He is an author or coauthor of 11 technical papers and a pending US patent.

Mahesh Kurapati, IBM, 200 West Madison Street, Chicago, IL 60606 (mkurapati@us.ibm.com).

Mr. Kurapati is an Advisory Software Engineer in the Business Analytics Department of the IBM Software Group. He received a B.E. degree in Electronics Engineering from Bangalore University, India, in 1993 and Specialization on P.C. Based Instrumentation from Indian Institute of Sciences, Bangalore, India. Before joining IBM in 2006 Mr. Kurapati was involved in various telecommunications and data mining projects. At IBM, Mr. Kurapati's primary focus is on the development of SPSS Statistics and SPSS Analytic Server products.