An Empirical Study on XML Schema Idiosyncrasies in Big Data Processing

Dmitry Vasilenko, Mahesh Kurapati

Business Analytics , IBM, Chicago, USA
{dvasilen, mkurapati}@us.ibm.com

Abstract

The design and maintenance of the XML schemas for the enterprise, if done incorrectly, can be difficult and frustrating experience. The applications that use XML data binding or participate in the processing of large and complex XML have to quickly adapt to support changing and evolving requirements without forcing each application that exchanges XML data or uses value-objects to upgrade in lock-step. While not ideally suited for the Big Data processing, XML remains the widely accepted data format of choice. This paper provides an overview of the practical techniques and approaches that can help an XML schema author to make intelligent design decisions.

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

I. INTRODUCTION

An XML designer, who does not have an extensive personal experience with the actual programming, will usually not have the intuition needed to make intelligent design decisions. An XML schema, while deemed to be correct and conformant might not be able to be transformed to value-objects for the particular programming language with the expected object layout or method signatures. Additionally, the processing of large and complex XML in map-reduce environments can be difficult and could require non trivial custom solutions. A number of generic solutions to processing XML for big data were proposed such as the one described in [1]. Such approaches rely on the fact that the XML data can be reliably and unambiguously transformed to predictable value-objects even if the XML schema for the underlying data is not readily available. The approaches discussed in this paper are derived in part from [2] and based on evidence and experience gained examining and working with the significant number of schemas for XML data binding and big data processing. The examples are illustrated by snippets from the XML schemas the authors worked with. The objective of the discussed approaches is not to dictate rules, but rather, to shed light on each design issue, so an informed decision can be made.

II. XML DATA BINDING

The W3C specification provides basic [3] and advanced [4] patterns known to be interoperable between state of the art data binding implementations. The examples shown below are derived from such patterns but mainly focus on eliminating any ambiguities on how the data binding can be achieved or interpreted even if the schema is not available to the implementation.

A. GLOBAL VERSUS LOCAL

The XML component (*element*, *complexType*, or *simpleType*) is "global" if it is an immediate child of the <schema> element, whereas it is "local" if it is nested within another component [5]. The [5] also provides a useful discussion on the design choices. The following example shows locally defined elements <freeForm> and <password> which, depending on the layout of the enclosing XML elements, can lead to ambiguities on how the names of the generated objects should be constructed.

```
<xs:choice>
 <xs:element name="freeForm">
  <xs:complexType>
   <xs:complexContent>
    <xs:extension base="baseType">
     <xs:attribute name="max" type="xs:int" use="optional"/>
     <xs:attribute name="min" type="xs:int" use="optional" default="0"/>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
```

```
 </xs:element>
 <xs:element name="password">
  <xs:complexType>
   <xs:complexContent>
    <xs:extension base="baseType">
     <xs:attribute name="max" type="xs:int" use="optional"/>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
 </xs:element>
</xs:choice>
```

On the other hand, defining the elements in the global scope eliminates any naming uncertainty:

```
<xs:element name="freeForm">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="baseType">
    <xs:attribute name="max" type="xs:int" use="optional"/>
    <xs:attribute name="min" type="xs:int" use="optional" default="0"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

<xs:element name="password">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="baseType">
    <xs:attribute name="max" type="xs:int" use="optional"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

.....
.....
.....

 <xs:choice>
  <xs:element ref="freeForm"/>
  <xs:element ref="password"/>
 </xs:choice>
```

This approach also correlates with the recommendations provided by Kohsuke Kawaguchi in [6].

B. UNNAMED ELEMENTS AND CARDINALITY

To support min/maxOccurs attributes for the unnamed XSD elements such as <choice> and <sequence> the XML binding frameworks tend to generate wrapper classes. The names of these wrapper classes can be awkward and inconsistent. Using named elements with min/maxOccurs attributes helps to avoid ambiguity. The following example shows the questionable usage of the min/maxOccurs for the <sequence> element:

```
<xs:element name="input">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="baseparameterType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="node"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
```

To support a such construct some code generators will generate a wrapper class, say, InputItem to wrap a Node element as shown in the example below:

```
public class Input extends com.example.BaseparameterType implements Serializable {
 public void addInputItem(com.example.InputItem vInputItem) throws IndexOutOfBoundsException {
  ......
 }
}

public class InputItem implements Serializable {
...
    private com.example.Node _node;
...
}
```

Such a class wrapper can be easily avoided by directly specifying the cardinality of the <node> element:

```
<xs:element name="input">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="baseparameterType">
    <xs:sequence>
     <xs:element ref="node" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
```

The same is true for the <choice> element as well.

C. ELEMENTS VERSUS TYPES

There are different schools of thought on how the types and elements should be expressed in the XML schema to achieve the desired interoperability and ease of maintenance. An overview of the "element vs. type"" considerations that also discusses the element nullability aspect is provided in [7]. Paul Downey in [8] provides a strong critique of the "xsi:type" usage.

The authors found that the following approach works the best:

1. Define a global type, e.g.:

```
<xs:complexType name="notificationElement" abstract="true">
  .....
</xs:complexType>
```

2. Derive a global element from the global type by extension:

```
<xs:element name="notificationDomain">
  <xs:complexType>
    <xs:complexContent>
          <xs:extension base="ns1:notificationElement">
                                    ....
</xs:element>
```

3. Use <element ref="..."> in the schema to refer to the element.

```
<xs:choice>
...
  <xs:element ref="ns1:notificationDomain" minOccurs="0"/>
...
</xs:choice>
```

This approach eliminates any variability on how the types and elements should be considered when dealing with the XML data binding. Admittedly, such an approach can also lead to designs that cannot easily evolve. As the terminal elements cannot be extended, the suggested design is mostly applicable to the well researched and modeled domains where the terminal nature of the resultant XML elements is acceptable.

D. PRIMITIVE TYPES AND OBJECT WRAPPERS

The XSD <choice> element allows only one of the elements contained in the <choice> declaration to be present within the containing element.

```
<xs:choice>
 <xs:element name="numberValue" type="xs:double"/>
 <xs:element name="dateValue" type="xs:date"/>
 <xs:element name="booleanValue" type="xs:boolean"/>
 <xs:element name="stringValue" type="xs:string"/>
</xs:choice>
```

While modern XML data binding implementations can handle a mix of the primitive types and object wrappers the resultant "choice" object can be difficult to process. The discussion about primitive vs. object dichotomy can be found in [9].

E. ELEMENT NAMES AND RESERVED WORDS

While mostly applicable to the older implementations, naming the XML elements using reserved words for a given programming language can lead to awkward and inconsistent method signatures. Consider the following XSD fragment that contains an element named "default":

```
<xs:complexType name="baseType">
 <xs:sequence>
  <xs:element name="description" type="xs:string"/>
  <xs:element name="default" type="xs:string" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

The generated method signatures did not follow the expected "camel" case [10] for the Java programming language and included underscores as shown below:

```
public void add_default(String inName) throws IndexOutOfBoundsException;
public void add_default(int i, String inName) throws IndexOutOfBoundsException;
public void clear_default();
```

On the other hand, the accessors for the simple "description" element have been generated correctly using normal Java naming conventions:

```
public void setDescription(String inName);
public String getDescription();
```

While the issue has been resolved in the later versions of the code generator, the clients were effectively locked into the erroneous API. The issue could be avoided by renaming the element to "defaultValue" so that the accessors have the expected method signatures.

F. EMBEDDED ENUMERATIONS

The generated name of an enumeration, defined as an immediate child of the attribute can contain the parent name, but this is not always desirable.

```
<xs:attribute name="visibility" use="optional" default="always">
 <xs:simpleType>
  <xs:restriction base="xs:string">
  <xs:enumeration value="noUI"/>
  <xs:enumeration value="hidden"/>
  <xs:enumeration value="always"/>
  </xs:restriction>
 </xs:simpleType>
</xs:attribute>
```

Defining the enumeration name in the global scope and referring to it using the type attribute as shown below helps to avoid ambiguity:

```
<xs:simpleType name="visibility">
 <xs:restriction base="xs:NMTOKEN">
  <xs:enumeration value="noUI"/>
  <xs:enumeration value="hidden"/>
  <xs:enumeration value="always"/>
  </xs:restriction>
</xs:simpleType>
......

...
<xs:attribute name="visibility" type="visibility" use="optional" default="always">
...
```

G. ELEMENT TYPE

Omitting the type of the XSD element will generate an object of anyType which should be avoided.

```
<xs:element name="fieldName"/>
```

Instead, the type of the element should be specified explicitly.

H. NAME AND REFERENCE

If the "name" is used in the XSD definition of an element or attribute the "type" should be explicitly specified. The following example shows an incorrect attribute specification that mixes the "name" and the "reference":

```
<xs:attribute name="hierarchy" ref="search:hierarchy" use="required"/>
```

The correct definition follows:

```
<xs:attribute name="hierarchy" type="search:hierarchy" use="required"/>
```

I. ARRAYS AND WRAPPERS

The authors observed that in some cases array wrapper elements such as "actionList", shown below, get filtered out by the code generator:

```
<xs:element name="actionList">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="search:actionSpecification" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

As a countermeasure one can use an additional "reserved" element, as shown below, to ensure that the wrapper does not get omitted:

```
<xs:element name="actionList">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="search:actionSpecification" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="reserved" type="xs:string" minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

III. XML SCHEMA AND BIG DATA PROCESSING

Efficient processing of XML in map-reduce environments can be rather challenging due to the "impedance mismatch inefficiencies" [11], size and complexity [12]. It is discussed in [1], that the Apache Hive XML serializer-deserializer (SerDe) can be successfully used to process the data of arbitrary complexity. Additionally, as was shown in [13], a software processor for Hive XML SerDe based on the Virtual Token Descriptor technology [14] can visibly improve performance.

The authors observed that XML processing for Big Data gravitates towards defining a minimal fragment of an XML document that can be reliably mapped to the Hive table row definition. The elements of such an XML fragment can be translated to the column definitions using appropriate XPath expressions. In a number of cases, however, the XML namespace declarations for the fragment will be lost as they are typically defined in the XML root element. The widely accepted workaround is to remove XML namespaces using XSLT transform before uploading the XML to the Hadoop file system. As XML schemas are not normally designed with such row-oriented metaphor in mind, the major difficulties are in finding such an XML fragment that will be self-sufficient and translatable to the desired row and column definitions. Additionally, complex data types often need to be flatten as shown in the following example.

Consider the following, simple and well designed, XML fragment:

```
<Report account="12345">
 <Session id="123" rid="ab123">
  <line by="z" name="pete"/>
  <line by="y" name="oxe"/>
 </Session>
 <Session id="456" rid="cd456">
  <line by="m" name="sam"/>
  <line by="n" name="doug"/>
 </Session>
 <Session id="789" rid="ef789">
  <line by="p" name="russ"/>
  <line by="q" name="john"/>
 </Session>
</Report>
```

that needs to be translated to the following output:

account	*session_id*	*reference_id*	*by*	*name*
12345	123	ab123	z	pete
12345	123	ab123	y	oxe
12345	456	cd456	m	sam
12345	456	cd456	n	doug
12345	789	ef789	p	russ
12345	789	ef789	q	john

The Hive table CREATE DDL for the above XML looks like this:

```
CREATE TABLE report(
        account string,
        sessions array<struct<id:string,rid:string,session:array<struct<by:string,name:string>>>>)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
        "xml.processor.class"="com.ximpleware.hive.serde2.xml.vtd.XmlProcessor",
        "column.xpath.account"="/Report/@account",
        "column.xpath.sessions"="/Report/Session"
) STORED AS
INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
TBLPROPERTIES (
        "xmlinput.start"="<Report",
        "xmlinput.end"="</Report>"
);
```

Note that the sessions column definition is somewhat complex even for such relatively simple XML.

To generate the desired output, two nested LATERAL VIEW transformations will be required, one to flatten the sessions and another to flatten the line array:

```
SELECT account, exp_sessions.id, exp_sessions.rid, line.by, line.name from
 (SELECT account,exp_sessions.exp_session.id,exp_sessions.exp_session.rid,      exp_sessions.exp_session
FROM   report
 LATERAL VIEW EXPLODE(sessions) exp_sessions as exp_session) exp_sessions
  LATERAL VIEW EXPLODE(exp_session.session) exp_lines as line;
```

A. ARRAYS AND WRAPPER ELEMENTS

Handling arrays without a wrapper element as in the example below is rather challenging if the parent element should be represented in a single Hive column.

```
<ReportingFI>
 <Name>a</Name>
 <Name>b</Name>
  <DocSpec>
    <DocRefId>d1</DocRefId>
  </DocSpec>
</ReportingFI>
```

A possible solution is to introduce a wrapper, Names, for the array so that the ReportingFI element can be represented as a structure that contains the array member. The modified example follows:

```
<FATCA_OECD version="1" schemaLocation="urn:oecd:ties:fatca:v1 FatcaXML_v1.1.xsd ">
 <MessageSpec>
  <MessageRefId>Sample1</MessageRefId>
 </MessageSpec>
 <FATCA>
  <ReportingFI>
   <Names>
    <Name>a</Name>
    <Name>b</Name>
   </Names>
    <DocSpec>
     <DocRefId>d1</DocRefId>
    </DocSpec>
  </ReportingFI>
  <ReportingFI>
   <Names>
    <Name>c</Name>
    <Name>d</Name>
   </Names>
    <DocSpec>
     <DocRefId>d2</DocRefId>
    </DocSpec>
  </ReportingFI>
 </FATCA>
</FATCA_OECD>
```

The CREATE TABLE DDL will look like this:

```
CREATE TABLE fatca(messagerefid string,
reportingfis
array<struct<reportingfi:struct<names:array<struct<name:string>>,docspec:struct<docrefid:string>>>>)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
 "xml.processor.class"="com.ximpleware.hive.serde2.xml.vtd.XmlProcessor",
 "column.xpath.messagerefid"="/FATCA_OECD/MessageSpec/MessageRefId/text()",
 "column.xpath.reportingfis"="/FATCA_OECD/FATCA/ReportingFI"
)
STORED AS
    INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
    TBLPROPERTIES (
    "xmlinput.start"="<FATCA_OECD ",
    "xmlinput.end"="</FATCA_OECD>"
    );
```

To flatten the data structure and generate the output one has to use the LATERAL VIEW:

```
set hive.exec.mode.local.auto=true;
set hive.cli.print.header=true;

SELECT messagerefid,
        exp_reportingfi.reportingfi.docspec.docrefid,
        exp_reportingfi.reportingfi.names
FROM fatca
LATERAL VIEW explode(reportingfis) exploded_reportingfis AS exp_reportingfi;
```

Running the SELECT query will produce the following output:

messagerefid	*docrefid*	*names*
Sample1	d1	[{"name":"a"},{"name":"b"}]
Sample1	d2	[{"name":"c"},{"name":"d"}]

B. HANDLING ENUMERATED TYPES

When dealing with the enumerations defined in such a way that the value of the type is referenced by an identifier, one can use the technique where the identifiers are collected into arrays, one per type as shown in the following DDL:

```
CREATE TABLE response_table(loan_id string, seller_number string,
    messages array<map<string,map<string,string>>>,
        product_ids array<string>, classification_ids array<string> )
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
 "xml.processor.class"="com.ximpleware.hive.serde2.xml.vtd.XmlProcessor",
 "column.xpath.loan_id"="//AttrValWrapper[AttrKey/text()='LoanId']/AttrVal/text()",
 "column.xpath.seller_number"="//AttrValWrapper[AttrKey/text()='SellerNumber']/AttrVal/text()",
 "column.xpath.messages"="//MessageWrapper",
 "column.xpath.product_ids" =
"//DecisionValWrapper[DecisionKey/text()='DecisionCategory'][DecisionVal/text()
        ='Product']/parent::DecisionValueContainer/parent::Decision/RowIdContainer//RowId/text()",
 "column.xpath.classification_ids" =
"//DecisionValWrapper[DecisionKey/text()='DecisionCategory'][DecisionVal/text()
        ='Classification']/parent::DecisionValueContainer/parent::Decision/RowIdContainer//RowId/text()"
)
STORED AS
   INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
   OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
   TBLPROPERTIES (
   "xmlinput.start"="<Answer>",
   "xmlinput.end"="</Answer>"
   );
```

To generate the enumerated value the SQL CASE statement with the array_contains() Hive UDF can be used to find if the collected array of identifiers indeed contains the identifier associated with the XML fragment being processed. The approach might also require flattening the structure using the LATERAL VIEW as shown below:

```
SELECT loan_id, seller_number,
CASE
 WHEN CAST(array_contains(product_ids, exp_message['MessageWrapper']['RowId'])
   AS STRING) = 'TRUE' THEN 'Product'
 WHEN CAST(array_contains(classification_ids, exp_message['MessageWrapper']['RowId'])
  AS STRING) = 'TRUE' THEN 'Classification'
 ELSE 'Unknown'
END AS category_code
FROM response_table
   LATERAL VIEW explode(messages) exploded_messages AS exp_message;
```

IV. CONCLUSION

In Big Data processing there is a tendency to represent the XML data using a relational metaphor that often conflicts with the commonly perceived tree-like nature of the XML. Implementing the translation layer to flatten the data after the XML schema is finalized can be rather challenging if such a usage of the XML was not considered during the design phase. The issues, techniques and approaches discussed in this paper can help designers to make intelligent decisions when developing schemas for XML that should be used for the data binding or processed in the map-reduce environments.

ACKNOWLEDGMENTS

Authors thank Vishwanath Kamat, Linda Liu, Vinayak Agrawal, Curtis Browning and Steven Halter of IBM for invaluable comments during preparation of this paper.

REFERENCES

[1] Dmitry Vasilenko, Mahesh Kurapati. Efficient Processing of XML Documents in Hadoop Map Reduce, IJCSE, 2014, Vol.6, No.9, p.329–333.
[2] XML Schemas: Best Practices, http://www.xfront.com/BestPracticesHomepage.html
[3] Basic XML Schema Patterns for Databinding Ver. 1.0, http://www.w3.org/2002/ws/databinding/edcopy/basic/basic.html
[4] Advanced XML Schema Patterns for Databinding Ver. 1.0, http://www.w3.org/2002/ws/databinding/edcopy/advanced/advanced.html
[5] Global versus Local, http://www.xfront.com/GlobalVersusLocal.pdf
[6] Kohsuke Kawaguchi, W3C XML Schema: DOs and DON'Ts, http://www.kohsuke.org/xmlschema/XMLSchemaDOsAndDONTs.html
[7] Should it be an Element or a Type? http://www.xfront.com/ElementVersusType.pdf
[8] Paul Downey, xsi:type is Evil, http://blog.whatfettle.com/2006/11/29/xsitype-is-evil/
[9] Sherman R. Alpert, Primitive Types Considered Harmful, http://www.research.ibm.com/people/a/alpert/ptch/ptch.html
[10] Camel Case, https://en.wikipedia.org/wiki/CamelCase
[11] Douglas Crockford, JSON: The Fat-Free Alternative to XML, http://www.json.org/fatfree.html
[12] Michael Driscoll, How XML Threatens Big Data, http://medriscoll.com/post/4741625281/how-xml-threatens-big-data
[13] Vinayak Agrawal, Processing XML data in BigInsights 3.0, https://developer.ibm.com/hadoop/blog/2014/10/31/processing-xml-data-biginsights-3-0/
[14] VTD-XML, https://en.wikipedia.org/wiki/VTD-XML