AVRO-457: Tools for Reading & Writing XML Files from/to Avro

This is a proposal for reading one or more XML files with the same XML Schema (XSD), and converting the contents of the XML schema into Avro data types, and writing the resulting Avro document. This proposal also covers translating data in the reverse direction.

Document Representation

XML Documents are a series of tags in tree form, starting with a root element that spans the entire document. Elements have attributes (named metadata whose values are only in text form), and both starting and ending tags. Between the starting and ending tags are one of: nothing (an empty tag), text, child elements, or a mixed combination of both text and child elements.

Child elements defined by an XML Schema are always part of some kind of group: *sequence*, *choice*, or *all*. In a *sequence* group, the child elements must occur in the order they are defined. In a *choice* group, one element must be present, but it can be any of the children defined in the choice group. Finally, all of the elements in an *all* group must appear, but they may occur in any order. Any group may contain sequence or choice sub-groups, but the all group must be top-level when used.

All child elements may be defined to occur between [0, unbounded] times. The default is for a child element to appear exactly once. This includes both the child elements themselves and groups containing child elements.

Finally, an element can define itself to be a substitution for another element. When this happens, the base element can be replaced by the substitution element wherever the base element appears in the document. For example, if element B declares itself to be a substitution of element A, element B may appear anywhere in the document that A is declared. This effectively declares A to be a choice group containing (A, B). *Note: A could also declare itself to be "abstract," meaning it can only be replaced by a substitution element. In the prior example, the choice group would contain only B.*

As a result, XML elements are best defined as Avro records. Each attribute is a key-value pair, whose value is a simple type, and likewise can be represented as a field with a primitive type. If an element is not empty, its content would be represented as a field with the same name as the element. Based on the element content, the value of this field would be:

- *Text-Only (Simple Content):* The value of that field would be the corresponding primitive type.
- *Elements-Only (Group):* Children can be represented as an array of union of all possible child types. This does not capture the fidelity of groups represented in XML Schema, but it is a decent format for reading the children.
- *Elements + Text (Mixed) Content:* This would be represented as an array of a union of string and the other types represented by child elements.

The last case to consider is where a child element may be substituted by another element. Since this is effectively a choice group, it is best represented as a union of all of the types that can represent that child element.

Simple XML Types vs. Avro Primitive Types

XML Schema defines a type hierarchy that far exceeds what can be represented in Avro. For example, there are 19 primitive data types in XML Schema compared to only 9 in Avro, including decimal. In XML Schema, all types are derived from anyType, and are expanded upon with the following different types of restrictions, called "facets:"

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- totalDigits
- fractionDigits

While most of these are self-explanatory, "pattern" is the most interesting. This defines a Perl regular expression used to describe the text. For example, the XML Schema integer type is restricted by a pattern facet of "[\-+]?[0-9]+".

As a result, all XML Schema simple types (defined in any schema, not just the base one) inherit from anyType with a set of facets constraining the text. A full hierarchy can be found here.

If the XML Schema reader follows the type system up the hierarchy, we only need to map specific Avro primitives to specific XML Schema simple types. The hierarchy will handle the rest. These are the proposed mappings for simple types:

- Avro *string* maps to XML Schema *anyType* as a catch-all.
- *bytes* maps to *base64Binary* and *hexBinary*.
- *float* maps to *float*.
- *double* maps to *double*.
- *decimal* maps to *decimal*.
- *long* maps to *long* and *unsignedInt*.
- *int* maps to *int* and *unsignedShort*.
- *boolean* maps to *boolean*.
- *null* has no mapping.

The facets that each derived type is restricted by has no effect when converting from XML to Avro. However, these facets will be validated against when converting Avro to XML.

Names and Namespaces

Like Avro complex types, elements and attributes in XML may have a namespace. In contrast, XML namespaces are URIs, while Avro namespaces are in the Java package-name style. I propose converting from XML namespaces to Avro namespaces under the following rules:

1. The Host name of the URI is split into its domain-name parts, and reversed.
2. The URI path is appended to the end, replacing "/" with "."

For example, the XML Schema namespace "`http://www.w3.org/2001/XMLSchema`" would become "`org.w3.www.2001.XMLSchema`" when generating an Avro record for an element in that namespace.

Optional Feature: Automatic Map Creation

XML Schema defines an ID attribute type, which "uniquely identifies the element that bears it." As such, if an element has an attribute of type "ID," a map can safely be generated with that attribute's value as the key and the element record as the value.

Note: Identity constraints also allow attributes to uniquely identify the elements that bear them. These define a uniqueness constraint on across a scope (a parent element), a selection of elements (defined by XPath), and a set of fields defining the unique key. This is much more difficult to properly represent in Avro, and I do not recommend trying to implement it.

Schema Conversion

The user may not want to transfer the entire XML document over to Avro. The user may wish to only transfer a simplified schema, or to select specific elements to encode in Avro.

When reading a document, no error will occur if a field name representing an attribute is missing from the Avro schema, or if not all possible child types are available in the child-element field. These elements and attributes will just be skipped over.

The user may also provide a schema containing only an array of a union of records representing the elements to transfer over. If so, the elements will be added to the array as they are encountered walking through the document, depth-first.

Schema Generation

It will be possible to generate an XML Schema from the Avro Schema, though with a lot of lost fidelity. The Avro primitive type mapping to XML Schema simple types would be:

- An Avro *string* maps to XML Schema's *anySimpleType*. (*anyType* is a complex type.)
- *bytes* maps to *base64Binary*.
- *float* maps to *float*.
- *decimal* maps to *decimal*.
- *double* maps to *double*.
- *long* maps to *long*.
- *int* maps to *int*.
- *boolean* maps to *boolean*.
- *null* has no mapping, and will be skipped.

Element children are all arrays of unions of types. If one of those types is a string, then the element has mixed content. The remaining records will be represented as a choice of elements defined to occur between 0 and unbounded times.

In the case of an Avro map generated from an ID field, there will be no way to reverse-engineer which record field represents the ID attribute. Likewise, the value of the map will be stored using the above rules, and the key will be skipped.

Addendum: Doug Cutting recommended storing XML Schema metadata inside the Avro schema to facilitate lossless conversion back. He provided the example: `{"type":"int", "xml-schema":"unsignedInt"}`

If a URL to the XML Schema was originally provided, we can encode that in the Avro schema metadata. If not, we can provide enough information in the Avro schema to generate an XML Schema that would validate all of the same documents that the original XML Schema would validate.

I propose to store group metadata as JSON objects, each of which with a "type" field containing the child type: “all,” “choice,” “sequence,” or “element.” Other fields define the minimum and maximum number of occurrences. For groups, a “value” field is an array of the members of that group. For elements, the “value” field is the element’s fully-qualified XML name.

Here is an example representation of a sequence that may occur at least 0 times and at most infinite, and contains only one element, which must occur exactly once.

```
{ "type": "sequence",
  "minOccurs": 0,
  "maxOccurs": "unbounded",
  "value": [
        { "type": "element",
          "minOccurs": 1,
          "maxOccurs": 1,
          "value": { "namespace": "http://www.w3.org/2001/XMLSchema",
                     "localPart": "complexType"
                   }
        }
  ]
}
```

Proposed Implementation

Dependencies

- *Apache XML Schema 2.1.0*: This handles the parsing of the XML Schema.
- *JRegex 1.2_01*: This handles regular expression validation when converting Avro to XML.

Components

XmlDatumWriter: Instance of DatumWriter<org.w3c.dom.Document>. If an XML Schema is either declared inside the document (via either the "schemaLocation" or "noNamespaceSchemaLocation" attributes), or provided in the XmlDatumWriter's constructor, that schema will be followed and compared to the output Avro schema. If the output schema cannot be supported using the rules from either the "Document Representation" section or the "Schema Conversion" section, an IOException will be thrown.

XmlDatumReader: Instance of DatumReader<org.w3c.dom.Document>. If an XML Schema is provided in its constructor, the Avro schema and data will be validated against it. If no XML Schema is provided, one can be automatically generated from the Avro schema via the rules in the "Schema Generation" section.

Utilities: Any further utility methods and classes required will be kept private to the package, only available to XmlDatumReader, XmlDatumWriter, and relevant unit tests.