# XML Comparison Project

Bertrand Benjamin

December 30, 2016

## Introduction

This assignment's aim at comparing the different techniques for parsing and processing XML. Code can be found at `https://github.com/Bertrandbenj/XML`

## SAX

Heavy weight method required for big files. SAX is an event based processing method, it allows you to skip every elements that you want and store whats needed the way you decide to keep it in memory. it is usually more efficient and therefore is the favoured method for read only requirements. It is the standard data processing method.

By overriding the startElement method we accumulate into a custom map the values that were not filtered. this solution will never outgrow the memory as it does increment the word's occurrence rather than counting them the memory complexity is linear to the number of distinct words.

```
public void startElement(String nameSpace, String localName, String qName, Attributes attr)
    if (qName.equals("w")) {
        String wrd = attr.getValue("lemma");
        if(wrd == null || wrd.length() < 3 || Common.isBlackListed(wrd))
            return;

        Map<String, Long> map = all.get(file.getName());
        map.put(wrd, map.getOrDefault(wrd,0L) + 1);
    }
}
```

In a second step, we build the String to be written to a file. this method could write the result map line by line rather than build a String if the memory requirements demands it but this would necessarily impact the file I/O that has to be synchronized each time [1] . In our case we do not have much memory constraint and therefore build a single String to have a single file write.

[1] `http://stackoverflow.com/questions/21632585/thread-safety-of-printstream-in-java`

```
all .entrySet()
    .stream()
    .map(ent -> "<document name=\"" + ent.getKey() + "\">\n    "
            + ent.getValue()
                .entrySet()
                .stream()
                .sorted((ent1,ent2)-> ent2.getValue().compareTo(ent1.getValue()))
                .filter(e -> e.getValue() > 3L)
                .map(m -> "<wrd cnt=\""+m.getValue()+"\" lemme=\""+m.getKey() +"\"/>")
                .collect(Collectors.joining("\n    "))
            + "\n  </document>"
        )
    .collect(Collectors.joining("\n  "));
```

# DOM

All in memory method, the good old implementation of the w3c specifications for hierarchic data structure. Its the basis for programmatic search of the tree structure. It suits read and write approaches. the entire parsing and serializing layer are handled... which is neat but it takes roughly 10 times more than the file size in memory, and require a full load of the file even to reach a single attribute. This can therefore seems overkill in many circumstances. In the real world this is reserved for configuration files and the like. Data processing may use it when there is a lot off updates to the tree (HTML page, web technologies).

In order to browse the tree, we use a TreeWalker with a NodeFilter that filters word Node and skip or ignore the rest also checking the word has a lemma attribute. We then iterate every node and accumulate a map similarly to the SAX implementation.

```
TreeWalker tw = td.createTreeWalker(input, NodeFilter.SHOW_ELEMENT,...

for (Node n = tw.nextNode(); n != null; n = tw.nextNode()) {
    String lemme = n.getAttributes().getNamedItem("lemma").getNodeValue();
    if(lemme.length()>2 && !Common.isBlackListed(lemme)){
        File ff = new File(n.getOwnerDocument().getBaseURI().substring(5));
        Map<String, Long> map = all.get(ff.getName());
        map.put(lemme, map.getOrDefault(lemme,0L)+1);
    }
}
```

DOM is an in-memory representation, it allows writes and tree creation. Rather than building a String output we build a XML tree and let the library handle the serialization.

```
static void mapToDoc(Document out ){
    all.forEach((doc,map) -> {
```

```
        Element d = out.createElement("document");
        d.setAttribute("name", doc);
        map.entrySet()
            .stream()
            .sorted((ent1,ent2)-> ent2.getValue().compareTo(ent1.getValue()))
            .filter(ent -> ent.getValue() > 3L)
            .forEach(ent -> {
                Element w = out.createElement("wrd");
                w.setAttribute("lemme", ent.getKey());
                w.setAttribute("cnt", ent.getValue()+"");
                d.appendChild(w);
            });
        out.getDocumentElement().appendChild(d);
    });
}
```

The serialization has a few parameters to be clean, have a DTD, a decent indentation

```
Transformer t = TransformerFactory.newInstance().newTransformer();
t.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, Common.DTD);
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
t.transform(new DOMSource(output), new StreamResult(new File("DOMOutput.xml")));
```

# XSLT

The XML fanatic method. when you really hate your colleagues or want to look cool. It's only used for templates and displays. XSLT is the typical language that aims at proving it is Turing complete and should have never left the lab. However, despite its unreadable appearance, XSLT possesses enough syntactic sugar to make it viable. Namely, the precious "xsl:for-each-group" , "xsl:analyze-string" and others. Associated with XPath, they allow aggregating filtering by regex.

```
java -jar saxon9he.jar corpus.xml ransform.xsl > XSLTOutput.xml
```

# JDOM

A more Java friendly implementation of DOM, JDOM has a few handy functions to access attributes and elements in a more readable way. It doesn't have fundamental memory or performance differences with DOM but is more java like. The demonstration of this integration is made using java 8 streams, in two steps. IN the first step we aggregate by documents, and by word which we count.

```
Stream.of(f)
    .peek(x -> System.out.println(x))
    .map(build)
    .map(doc ->  doc.getRootElement())
    .flatMap(s -> recursiveSearch(s))
    .filter(s -> s.getName().equals("w") )
    .filter(s -> s.getAttributeValue("lemma") != null)
    .filter(s -> s.getAttributeValue("lemma").length() > 2  )
    .filter(s -> !Common.isBlackListed(s.getAttributeValue("lemma") ))
    .collect(groupingBy(
                e -> new File(e.getDocument().getBaseURI()).getName(),
                groupingBy(
                    wrd -> wrd.getAttributeValue("lemma"),
                    counting())
            ));
```

In the second part we sort the hashmap's entry in order to aggregate them back into a single element by successive reduce. This reproduce a tree by creating Element from the leaf on to the root.

```
.map(ent -> ent .getValue()
                .entrySet()
                .stream()
                .sorted((ent1, ent2) -> ent2.getValue().compareTo(ent1.getValue()))
                .filter(e -> e.getValue() > 3)
                .reduce(
                        new Element("document").setAttribute("name", ent.getKey()),
                        (t,u) -> {
                            if(t != null){
                                Element wrd = new Element("wrd")
                                    .setAttribute("cnt", u.getValue().toString())
                                    .setAttribute("lemme", u.getKey());
                                return  t.addContent(wrd);
                            }
                            return t;
                        }
                        ,(t,u) -> t.addContent(u)
                ))
.reduce(new Element("Root"),
        (t,u) -> u==null?t:t.addContent(u));
```

We have therefore pipped the entire execution into map and reduce component. those describe the "What" we want to achieve and not the "How". Indeed, the VM will be capable of adapting the code, reorder the components to achieve the best possible performance.

## XPath

Using XPath, the document is loaded in a DOM tree and then walked through using a XPathExpression. This means we have two overhead, the DOM loading and the XPath engine. it can be very useful when building advanced query like database kind of selects. however XPath isn't suited to do aggregates and transformations. Its a query language and all it helps us with here is the filtering of node similar the the Tree-Walker of the DOM method.

```
XPathExpression expr = xpath.compile("//w[@lemma
            and not(contains('"+Common.simpleWords+"', @lemma))
            and string-length(@lemma) > 2]");
```

## Apache Spark

Many more option can be thought of to implement the same algorithm yet another time. Spark allows distribution of the data meaning we can store an XML in memory in a distributed fashion together with its schema. This way we have a raw file and its DTD floating around a cluster of server. The point ? Having an in memory document that is roughly the size of the input file (may even be smaller) and a schema (similar to a dtd) in order to access any element. its very heavy weight, read only system but can take in terabytes of hierarchic data (millions of millions of XML fragments). A capability no other system can.

## DTD

```
<!ELEMENT Root (document*)>
<!ELEMENT document (wrd*)>
<!ELEMENT wrd EMPTY>
<!ATTLIST wrd cnt CDATA #REQUIRED>
<!ATTLIST wrd lemme CDATA #REQUIRED>
<!ATTLIST document name CDATA #REQUIRED>
```

## Conclusion

XML has many libraries to transform, store, manipulate data. Each have its advantages and inconvenient. We have seen a few here but could also mention JAXB to map java objects to the XML tree and allow on the fly serializations. Globally, two types are distinguished, in memory trees (DOM) and file parsers (SAX). Spark would represent an "in between" solution but won't be detailed here.

The main questions to asks before choosing a solution or another are :

- What size is the input?

- Do I need to access many different nodes?
- Do I need recurrent accesses?
- Am do I need to change / update the tree often?
- Is a tree structure more suitable ?

Finally, the execution are not surprising, SAX is the fastest with an average of 700ms, then comes DOM and JDOM with 1500ms (DOM seems to have more consistent execution time). With roughly 2000ms The XPath version takes the 4th spot as it add an extra layer of code. And the palm goes to XSLT that add a third layer overhead: parsing and building the XML code, it surprisingly takes barely more than 2 seconds and so I suspect the execution may actually be quite efficient once we forget the overheads.