XML Processing in Scala

Dino Fancellu
Felstar Ltd
<dino@felstar.com>

William Narmontas
Apt Elements Ltd
<william@scalawilliam.com>

Abstract

Scala is an established static- and strongly-typed functional and object-oriented scalable programming language for the JVM with seamless Java interoperation. Scala and its ecosystem are used at LinkedIn, Twitter, Morgan Stanley among many companies demanding remarkable time to market, robustness, high performance and scalability.

This paper shows you Scala’s strong native XML support, powerful XQuery-like constructs, hybrid processing via XQuery for Scala, and increased XML processing performance. You will learn how you can benefit from Scala’s practicality in a commercial setting, ultimately increasing your productivity.

Keywords: Scala, XML, XQuery, XSLT, XQJ, Java, Processing

1. Introduction

Programming style: Scala’s immutability, functional programming, first-class XML make it rather similar to XQuery. Scala’s for-comprehensions were inspired by Philip Wadler from his work with XQuery. [1]

Ecosystem: Scala’s seamless Java interoperation gives you access to all of Java’s libraries, the JVM [2] and many outstanding Scala libraries 1 2 3.

Scalability: Scala’s scalability and design negate the need for design patterns in solving a language’s design flaws. It is everything that Java should have been.

XML handling: Scala’s XML handling includes the standard XML types such as Element, Attribute, Node. It also includes the NodeSeq type which extends Seq[Node] (a sequence of nodes), meaning that all of Scala’s collections functionality for sequences is available for XML types. The key Scala XML documentation can be found at its author’s Burak Emir’s Scala XML book [3], scala.xml API 4 and scala-xml GitHub repository 5.

2. Five minutes to understanding Scala

This paper covers a relevant selection of Scala’s capabilities. There are many great resources to learn about traits, partial functions, case classes, etc. We will cover the necessary essentials for this paper. See Scala crash course [4] and a selected presentation [5] for detailed walk-throughs.

Like with XQuery and other functional programming languages we recommend programming Scala in an immutable fashion, although Scala allows you to program in an Object Oriented fashion or hybrid of the two, making it especially suited to migrating from a Java code base.

Scala’s types are static, strong and mostly inferred, to the extent that it can feel like a scripting language [6]. Your IDE and Scala’s compiler will inform you of your program’s correctness very early on - including XML well-formedness.

Scala’s ‘implicits’ enable you to define new methods on values in a limited scope. With implicits and type inference your code becomes very compact [7] [8]. In fact, this paper displays types only for the sake of clarity.

---

1 ScalaTest - http://scalatest.org
2 Akka - http://akka.io/
3 Play Framework - http://www.playframework.com/
5 scala-xml GitHub repository - https://github.com/scala/scala-xml/
Scala is about expressions, not statements. The last expression in a block of expressions is the return value. The same applies to if-statements and try-catch.

Scala is best used from within IntelliJ IDEA and Eclipse with the Scala IDE plug-in. [9]

### 2.1. Values and functions

Scala & XQuery:
- def fun(params): type similar to declare function local:fun(params): type
- val xyz = (expression) similar to let $xyz := (expression)

Functions can be passed around easily. Example:

```scala
def incrementedByOne(x: Int) = x + 1
(1 to 5).map(x => x + 1)
```

This example however can be slimmed down to

```scala
(1 to 5).map(x => x + 1)
```

Where `x => x + 1` is an anonymous (lambda) function. It can be slimmed down further to

```scala
(1 to 5).map(_ + 1)
```

Scala’s collections, such as lists, sets and maps come in mutable and immutable flavours [10]. They will be used throughout the examples.

### 2.2. Strings and string interpolation

The triple double-quote syntax negates escaping of double-quotes in string literals. E.g.

```scala
val title = """An introduction to "Scala"""
```

Scala supports string interpolation [11] similar to that in PHP, Perl and CoffeeScript - with the ’s’ modifier:

```scala
val language = "Scala"
val interpolatedTitle = s""""An introduction to "$language"""
```

String interpolation turns $language into $language.toString.

Scala’s triple-quoted strings may be multi-line, as shown in the examples section.

### 2.3. Named parameters

Where further clarity for method calls is needed, you can use named parameters:

```scala
def makesLink(url: String, text: String) = s""""<a href='$url'>$text</a>"
```

```scala
makesLink(text = "XML London 2014",
url = "http://www.xmllondon.com/")
```

```xml
<a href='http://www.xmllondon.com/'>XML London 2014</a>
```

### 2.4. For-comprehensions

For-comprehensions [12] will be familiar to a programmer who has used Python, LINQ, XQuery, Ruby, Haskell, F#, Erlang, Clojure.

You can rewrite the previous example

```scala
(1 to 5).map(x => x + 1)
```

as a for-comprehension:

```scala
for {x <- (1 to 5) } yield x + 1
```

These comprehensions yield results by iterating over multiple collections:

```scala
val software = Map(
  "Browser" -> Set("Firefox", "Chrome", "Internet Explorer"),
  "Office Suite" -> Set("Google Drive", "Microsoft Office", "Libre Office")
)
for { (softwareKind, programs) <- software 
  program <- programs 
  if program.endsWith "e"
} yield s"$softwareKind: $program"
```

```scala
List(Browser: Chrome, Office Suite: Google Drive, Office Suite: Microsoft Office, Office Suite: Libre Office)
```

Inside a for-comprehension, Scala and XQuery once again share similarities:

- x <- (expression) similar to
  - for $x in {expression}
- if (condition) similar to
  - where {condition}
- abc = (expression) similar to
  - let $abc := {expression}
- yield (expression) similar to
  - return {expression}

XML Processing in Scala
3. Scala's strong native XML support

Unlike in Java, XML is a first class citizen in Scala and can be used as a native data type.

The scala.xml library source code is available on GitHub.1

3.1. Basic Inline XML

XML literals can be embedded directly in code with curly braces.

```scala
val title = "XML London 2014"
val xmlTree =
  <div>
    <p>Welcome to <em>{title}</em>!</p>
  </div>
```

Serializing this XML structure works as expected:

```scala
xmlTree.toString
```

These XML literals are checked for well formedness at compile time or even in your IDE reducing errors.

Curly braces can be escaped with double braces. e.g.

```scala
val squiggles = <root>I like {{squiggles}}</root>
```

3.2. Reading

Scala can load XML from Java’s File, InputStream, Reader, String using the scala.xml.XML object. Here is an XML document in String form:

```scala
val pun =
  """<pun rating="extreme">
  | <question>Why do CompSci students need glasses?</question>
  | <answer>To C#!</answer>
  | C# is a Microsoft's programming language
  | ...</answer>
  | </pun>
```

Loading an XML document from a String gives us a node:

```scala
scala.xml.XML.loadString(pun)
```

When you need XML comments use the ConstructingParser [13]:

```scala
scala.xml.parsing.ConstructingParser
  .fromSource(scala.io.Source.fromString(pun),
  preserveWS = true).document().docElem
```

```scala
<pun rating="extreme">
  <question>Why do CompSci students need glasses?</question>
  <answer>To C#!</answer>
  C# is a Microsoft's programming language
  ...</answer>
</pun>
```

3.2.1. Look ups and XPath alternatives

Scala has its own XPath-like methods for querying from XML trees.

```scala
val listOfPeople = <people>
  <person>Fred</person>
  <person>Ron</person>
  <person>Nigel</person>
</people>
```

```scala
listOfPeople \ "person"
```

```scala
NodeSeq(<person>Fred</person>,
  <person>Ron</person>, <person>Nigel</person>)
```

Looking for descendants

```scala
val fact = <fact type="universal">
  <variable>A</variable>
</fact>
```

```scala
fact \ "variable"
```

```scala
NodeSeq(<variable>A</variable>,
  <variable>A</variable>)
```

Querying attributes is similar

```scala
fact \ @"type"
```

```scala
i: scala.xml.NodeSeq = universal
```

```scala
fact @"type"
```

```scala
i: String = universal
```

---

1 scala.xml library GitHub - https://github.com/scala/scala-xml/
Looking up elements by namespace (see Appendix A, `The showNamespace(-s)` methods for showNamespaces):

```scala
val tree = <document
  <embedded xmlns="urn:test:embedding">
    <description>
      <referenced xmlns="urn:test:referencing">
        <metadata>
          <title xmlns=""">Untitled</title>
        </metadata>
      </referenced>
    </description>
  </embedded>
</document>
```

Looking up elements by namespace (see Appendix A, `The showNamespace(-s)` methods for showNamespaces):

```scala
(tree \ "._").filter(_.namespace == "urn:test:referencing").map(showNamespace).foreach(println

{urn:test:referencing}referenced
{urn:test:referencing}metadata
```

Looking up attributes by namespace:

```xml
<node xmlns="urn:meta" demo="test"/>

<node xmlns:meta="urn:meta" meta:demo="test"/>
```

The reason that backslashes were chosen instead of the usual forward slashes is due to the use of // for Scala comments, i.e. the // would never even be seen.

Scala’s XML is displayed as a NodeSeq type which extends `Seq[Node]`. This means we get Scala’s collections for free. Here are some examples:

```scala
val root = <numbers>
  for {i <- 1 to 10} yield
  <number>(i)
</numbers>
```

```scala
val numbers = root \ "number"
nodes(8)
```

```scala
numbers.head
```

```scala
numbers.last
```

```scala
numbers take 3
```

```scala
numbers filter(_.text.toInt > 6)
```

```scala
val jokes = <jokes>
  <pun rating="fine">
    <question>Q: Why did the functions stop calling each other?</question>
    <answer>A: Because they had constant arguments.</answer>
  </pun>
  <pun rating="extreme">
    <question>Why do CompSci students need glasses?</question>
    <answer>To C#</answer>
    C# is a Microsoft programming language
    --
    <answer>
  </pun>
</jokes>
```

```scala
val numbers = numbers.maxBy(_.text.toInt)
```

```scala
val numbers = numbers.maxBy(_.text.toInt)
```

```scala
val numbers = numbers.reverse
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```

```scala
val numbers = numbers.groupBy(_.text.toInt % 3)
```
Querying attributes by path:

```scala
document 
  <jokes> 
    <pun> 
      flatMap (_ \ "@rating") 
    </pun> 
  </jokes>
```

NodeSeq(fine, extreme)

However node equality can surprise with XML literals 1:

```scala
<node>{2}</node> == <node-2</node>
false
<node>{2}</node> == <node>{2}</node>
true
```

3.3. Scala XML namespace handling

Namespaces are handled well. The empty namespace is 'null'. (see Appendix A, The showNamespaces(-s) methods for showNamespaces):

```scala
val tree = <document>
  <jokes>
    <pun> 
      flatMap (_ \ "@rating") 
    </pun> 
  </jokes>
</document>
```

showNamespaces(tree)

```scala
{urn:test:embedding}embedded
{urn:test:embedding}description
{urn:test:referencing}referenced
{urn:test:referencing}metadata
{null}title
```

3.3.1. Scala XML is unidirectional and immutable

Unlike the XPath model, Scala XML is unidirectional, i.e. a node does not know its parent, so lacks reverse axes, also no forward/sibling axes. This was done because adding in parents is expensive whilst maintaining immutability. For many problem spaces that may not matter. If it does for you then you are free to fall back to the full XPath/XQuery/XSLT model as shown below

3.3.2. XQS

We use a tiny wrapper library called XQS (XQuery for Scala) [14] in various places throughout this paper. Its main aim is to allow for a Scala metaphors when using XQuery. However even outside of XQuery usage, it allows for easy interoperability between the worlds of Scala XML and Java DOM. For example in the XPath example below, it supplies toDom to turn Scala XML to a w3c DOM, and the ability to turn a NodeSet into a Scala NodeSeq.

3.3.3. Using XPath from Scala

```scala
import com.felstar.xqs.XQS._
val widgets = <widgets>
  <widget>Menu</widget>
  <widget>Status bar</widget>
</widgets>
```

val xpath = XPathFactory.newInstance().newXPath()
val nodes: NodeSeq = xpath.evaluate("/widgets/widget[not(@id)]",
toDom(widgets), XPathConstants.NODESET).asInstanceOf[NodeList]
nodes

Natively in Scala:

```scala
(widgets \ "widget")(
  widget => (widget \ "@id").isEmpty
)
```

1 https://github.com/scala/scala-xml/issues/25
3.3.4. XML Transformations

Scala provides XML transformation functionality via a `RuleTransformer` that takes multiple `RewriteRule`s. The following example uses pattern matching and a native XML extractor:

```scala
val peopleXml = <people>
  <john>Hello, John.</john>
  <smith>Smith is here.</smith>
  <another>Hello.</another>
</people>

val rewrite = new RuleTransformer(new RewriteRule {
  override def transform(node: Node) =
  node match {
    case <john>{_}</john> =>
      <john>Hello, John.</john>
    case <smith>({text}</smith>) =>
      <smithX>{text}!!!!</smithX>
    case n: Elem if n.label != "people" =>
      n.copy(label = "renamed")
    case other => other
  }
})
rewrite.transform(peopleXml)
```

Alternatively: calling XSLT from Scala

```scala
val stylesheet = <xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="2.0">
  <xsl:template match="john">
    <xsl:copy>Hello, John.</xsl:copy>
  </xsl:template>
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>

import com.felstar.xqs.XQS._
val xmlResultResource = new java.io.StringWriter()
val xmlTransformer = TransformerFactory.newInstance().newTransformer(stylesheet)
xmlTransformer.transform(peopleXml,
  new StreamResult(xmlResultResource))
xmlResultResource.getBuffer
```

We found XSLT more effective than Scala for designing XML transformations as XSLT has been designed explicitly for this task. Thus we can mix-and-match transformations when XSLT is nicer than Scala and vice-versa. John Snelson’s transform.xq showcases mixed transforms with querying in XQuery [15]. Alike can be achieved in Scala.
3.3.5. XML Pull Parsing from Scala

```scala
// 4GB file, comes back in a second.
val downloadUrl = "http://dumps.wikimedia.org" + "/enwiki/20140402/enwiki-20140402-abstract.xml"
val src = Source.fromURL(downloadUrl)
val er = XMLInputFactory.newInstance().createXMLEventReader(src.reader)
implicit class XMLEventIterator(ev: XMLEventReader) extends scala.collection.Iterator[XMLEvent] {
  def hasNext = ev.hasNext
  def next = ev.nextEvent()
}
er.dropWhile(!_isStartElement).take(10).zipWithIndex.foreach {
  case (ev, idx) => println(s"${idx+1}:	$ev")
}
src.close()
```

3.3.6. Calling XQuery from Scala

The standard API for XQuery on Java is XQJ [16]. XQJ drivers are available for several databases such as
MarkLogic and XQuery processors such as Saxon [17] [18] meaning Scala can consume XQuery result sets.

```scala
import com.felstar.xqs.XQS._
val conn = getYourXQueryConnection()
val ret: NodeSeq = conn("/widgets/widget[not(@id)]", widgets)
   NodeSeq(<widget>Menu</widget>, <widget>Status bar</widget>)
val ret2: NodeSeq = conn(""|for $w in /widgets/widget
  |order by $w return $w"".stripMargin, widgets)
   NodeSeq(<widget>Menu</widget>, <widget id="panel-1">Panel</widget>,<widget id="panel-2">Panel</widget>,<widget>Status bar</widget>)
```

A pure Scala version:

```scala
(widgets \ "widget").sortBy(_.text)
```

4. Extensibility

Using Scala’s “implicits” you can enrich types by adding new functionality.

```scala
val oo = <oo>
  <x id="1">123</x>
  <x id="2">1234</x>
  <x id="x">xxxxx</x>
  <x id="3">1235</x>
</oo>
```

Treating attribute values, which are strings, as doubles, implicitly when needed, and without any
NumberFormatExceptions. Uses the scala.util.Try class that wraps exceptions in a functional manner

```scala
implicit def toSafeDouble(st: String) = scala.util.Try(st.toDouble).getOrElse(Double.NaN)
```

```scala
(oo \ "x").filter(_\@"id" < 3)
```

```scala
NodeSeq(<x id="1">123</x>, <x id="2">1234</x>)
```

```scala
(oo \ "@id").map(_.text: Double).filterNot(_.isNaN).sum
```

Here are some examples of selecting multiple items according to their index:

```scala
def filterByIndex(p: Int => Boolean): NodeSeq
  { ns.zipWithIndex.collect
    case (value, index) if p(index) => value
  }

val root = <nodes>
  <node>a (0)</node>
  <node>b (1)</node>
  <node>c (2)</node>
  <node>d (3)</node>
  <node>e (4)</node>
  <node>f (5)</node>
  <node>g (6)</node>
  <node>h (7)</node>
  <node>i (8)</node>
</nodes>
val ns = (root \ "node")

implicit class indexFunctionality(ns: NodeSeq) {
  def filterByIndex(p: Int => Boolean): NodeSeq = ns.zipWithIndex
    .collect { case (value, index) if p(index) => value
  }
  def apply(n1: Int, n2: Int*) = ns(n1) ++ ns.filterByIndex(n2).toSet
  def apply(b: GenSeq[Int]*): NodeSeq = filterByIndex(b).flatten.toSet
}

nodes.filterByIndex(_ > 6)

NodeSeq(<node>a (0)</node>, <node>e (4)</node>, <node>h (7)</node>)
nodes(0, 4, 7)

NodeSeq(<node>a (0)</node>, <node>e (4)</node>, <node>h (7)</node>)
nodes(1 to 3)

NodeSeq(<node>b (1)</node>, <node>c (2)</node>, <node>d (3)</node>)
nodes(1 to 3, 5 until 7)

NodeSeq(<node>b (1)</node>, <node>c (2)</node>, <node>d (3)</node>, <node>f (5)</node>, <node>g (6)</node>)
```

Note that root can alternatively be generated using:

```scala
val root = <nodes>('a' to 'i').zipWithIndex.map{
  case (letter, index) =>
    <node>[letter] {[index]}</node>
}</nodes>
```

This is how we look up elements by namespace. You can see how extensible Scala becomes (using Appendix A, The showNamespace(-s) method):

```scala
val tree = <document>
  <embedded xmlns="urn:test:embedding">
    <description>
      <referred xmlns="urn:test:referencing">
        <metadata>
          <title xmlns="">Untitled</title>
        </metadata>
      </referred>
    </description>
  </embedded>
</document>

implicit class nsElement(nodeSeq: NodeSeq) {
  val regex = "^\{(.+)\}(.+)$".r
  def \#(path: String): NodeSeq = {
    val regex(namespace, el) = path
    for {
      node <- nodeSeq \ el
      if node.namespace == namespace
    } yield node
  }
  def \#(path: String): NodeSeq = {
    val regex(namespace, el) = path
    for {
      node <- nodeSeq \ el
      if node.namespace == namespace
    } yield node
  }
}
```

```
{urn:test:referencing}referred
{urn:test:referencing}metadata
```

(see Appendix A, The showNamespace(-s) method)
4.1. Further Extensibility: XQuery-like constructs

Here we implement the XQuery 3.0 use case Q4 Part 3 [19].

XQuery code:

```
<result>
  for $store in /root/*/store
  let $state := $store/state
  group by $state
  order by $state
  return <state name="{$state}">
    for $product in /root/*/product
    let $category := $product/category
    group by $category
    order by $category
    return <category name="{$category}">
      for $sales in /root/*/record
      store-number = $store/store-number
      and product-name = $product/name
      let $name := $sales/product-name
      group by $name
      order by $name
      <product name="{$name}"
      total-qty="{$sum($sales/qty)}"/>
    </category>
  </state>
</result>
```

Scala code:

```scala
def loadXML(ref: String) = {
  val filename = s"benchmarks-xml/$ref"
  val file = new File(filename)
  scala.xml.XML.loadFile(file)
}

val allStores = loadXML("stores.xml") \ "store" groupByOrderBy "state"
val allProducts = loadXML("products.xml") \ "product" groupByOrderBy "category"
val allRecords = loadXML("sales-records.xml") \ "record" groupByText "product-name"

<result>
  for {
    (state, stateStores) <- allStores
    storeNumbers = (stateStores \ "store-number").textSet
  } yield <state name="{$state}">
    for {
      (category, products) <- allProducts
      productRecords = allRecords.filterKeys((products."name").textSet)
    } yield <category name="{$category}">
      for {
        (productName, productSales) <- productRecords
        filteredSales = productSales.filter(n => storeNumbers(n."store-number" text) )
        if filteredSales.nonEmpty
        totalQty = (filteredSales \ "qty").map(_.text.toInt).sum
      } yield <product name="{$productName}"
        total-qty="{$totalQty.tostring}"/>
    </category>
  </state>
</result>
```

An extensibility class used is attached in Appendix B, Extensions for NodeSeq.
5. Performance vs XQuery

5.1. Assumptions

Core i7-3820 @ 3.6 GHz, 4 core, Windows 7 Professional, 64 bit, 16 GB Ram, Java 7 u51 64bit, default JVM settings. Scala 2.11, XMLUnit, XQJ interfaces, XQS Scala bindings 2 XQuery implementations A and B. Sources are located on GitHub [20].

5.2. Methodology

Using prepared statements for the XQuery, can be switched off, B performance drops like a stone without it, and not really fair, so turn on prepared statements. Scala has no concept of these, as there is nothing to prepare or parse. Also cached the conversion of XML to a DOMSource for the XQuery, so we don’t measure that effort when timing the XQueries. Put in switch to serialize results to string, so as to ensure that any potential lazy values are materialized. Selected various queries from [21] also a XQuery 3.0 example from [19]. Runs both XQuery and Scala in a single run, 3 runs of 10,000 queries, with the results of the first 2 runs thrown away to get a good JVM jit warmup. Its very easy to get misleading results from badly thought out benchmarks. Warm up is very important, JVM runs best when code is hotspotted. For each query we emit the XQuery time, Scala time, and the ratio of these times, XQuery:Scala. We plot a graph of these values, showing first 2 values as a bar, the ratio as a line.

5.3. Benchmarks

<table>
<thead>
<tr>
<th>Query</th>
<th>Ratio</th>
<th>XQuery</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>38.27</td>
<td>2449</td>
<td>64</td>
</tr>
<tr>
<td>Q2</td>
<td>29.89</td>
<td>2600</td>
<td>87</td>
</tr>
<tr>
<td>Q3</td>
<td>33</td>
<td>2574</td>
<td>78</td>
</tr>
<tr>
<td>Q4</td>
<td>7.75</td>
<td>3325</td>
<td>429</td>
</tr>
<tr>
<td>Q5</td>
<td>23.91</td>
<td>3372</td>
<td>141</td>
</tr>
<tr>
<td>Q6</td>
<td>40.1</td>
<td>2927</td>
<td>73</td>
</tr>
<tr>
<td>Q7</td>
<td>17.86</td>
<td>2590</td>
<td>145</td>
</tr>
<tr>
<td>Q9</td>
<td>32.04</td>
<td>1602</td>
<td>50</td>
</tr>
<tr>
<td>Q10</td>
<td>12.48</td>
<td>2994</td>
<td>240</td>
</tr>
<tr>
<td>Q11</td>
<td>26.86</td>
<td>2847</td>
<td>106</td>
</tr>
<tr>
<td>Q4_3.0</td>
<td>6.89</td>
<td>4921</td>
<td>714</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Ratio</th>
<th>XQuery</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>9.57</td>
<td>603</td>
<td>63</td>
</tr>
<tr>
<td>Q2</td>
<td>7.11</td>
<td>619</td>
<td>87</td>
</tr>
<tr>
<td>Q3</td>
<td>7.4</td>
<td>592</td>
<td>80</td>
</tr>
<tr>
<td>Q4</td>
<td>1.74</td>
<td>750</td>
<td>430</td>
</tr>
<tr>
<td>Q5</td>
<td>5.44</td>
<td>816</td>
<td>150</td>
</tr>
<tr>
<td>Q6</td>
<td>8.26</td>
<td>611</td>
<td>74</td>
</tr>
<tr>
<td>Q7</td>
<td>3.89</td>
<td>579</td>
<td>149</td>
</tr>
<tr>
<td>Q9</td>
<td>4.59</td>
<td>225</td>
<td>49</td>
</tr>
<tr>
<td>Q10</td>
<td>2.21</td>
<td>478</td>
<td>216</td>
</tr>
<tr>
<td>Q11</td>
<td>6.33</td>
<td>658</td>
<td>104</td>
</tr>
<tr>
<td>Q4_3.0</td>
<td>2.49</td>
<td>1715</td>
<td>688</td>
</tr>
</tbody>
</table>
5.4. Conclusions

Scala is faster in all these use cases. Very similar to XQuery in its language construction. No doubt there are use cases where XQuery may be better, like an XML database. This is not black or white, a religious issue, simply a matter of choice.

6. Practicality

6.1. Enterprise usage

Scala is well established in enterprises [22]. While having access to the JVM Scala makes it easy to reuse the solid and tested libraries of the JVM ecosystem as well as an enterprise’s legacy Java code [23]. Scala’s terseness makes domain modelling much more precise [24]. Enterprise can migrate slowly to using all-Scala. The amount of code to maintain decreases, so number of moving parts decreases.

6.2. ScalaTest

ScalaTest, then test either your whole domain with property based testing, and ensure that the parties you are dealing with understand what your XML processing code does. Again, whether your XML processing code is inside Scala, XSLT, XQuery or MarkLogic, makes no difference. XMLUnit works nicely with Scala.

6.3. Other integration features

Scala 2.11 makes itself available as a scripted language to JSR-223 [25]. Scala’s Akka 1 and 2 provide many integration features with the rest of the world including JSON 3 and WebSockets 4. With macros you can create programs that create programs. Meaning your language is not getting in your way with ‘design patterns’ when focusing on the problem you’re trying to solve. This includes creating bindings such as serializers and deserializers of your favourite formats (e.g. binary via Scala Pickling 5, JSON via json4s 6.

We would like to see more research in querying with Scala such as Fatemeh Borran-Dejnabadi’s paper [26].

7. Conclusions

Possibilities with using Scala for XML processing are almost limitless. Pick and mix how you want to process your XML in Scala: powerful collections methods, for-comprehensions, XML generation, XPath, XSLT, XML databases and XQuery engines via XQ5/XQJ, XML streaming via StAX. Scala makes it possible to simplify complex logic into domain specific programs and use a combination of the best tools for achieving your targets. As Java has not advanced as far in terms of the language, Scala has secured the niche of the effective programmer and the effective business. For you as a functional programmer Scala’s concepts will already be familiar. You lose none of your existing Java ecosystem and gain so much more. It is another important tool in your armoury for efficient and lucid data processing.

---

1 Akka - http://akka.io/
2 Play framework - http://www.playframework.com/
3 http://www.playframework.com/documentation/2.2.x/ScalaJson
4 Play Framework documentation, WebSockets in Scala guide - http://www.playframework.com/documentation/2.2.x/ScalaWebSockets
5 http://lampwww.epfl.ch/~hmiller/pickling/
6 http://json4s.org/
Bibliography

Sadek Drobi and Martin Odersky. InfoQ.
http://www.scala-lang.org/what-is-scala.html#seamless_java_interop
http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/00-crash.html
http://www.artima.com/weblogs/viewpost.jsp?thread=328540
http://www.artima.com/weblogs/viewpost.jsp?thread=179766
http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/01-iterators.slides.html
https://jcp.org/en/jsr/detail?id=225
http://www.w3.org/TR/xquery-30-use-cases/#groupby_q4
http://infoscience.epfl.ch/record/85493/files/Scala_XQuery.pdf
A. The showNamespace(-s) methods

```scala
def showNamespace(node: Node) = s"${node.namespace}\${node.label}"

def showNamespaces(ofTree: NodeSeq) =
  (ofTree \ " ").map(showNamespace).mkString("\n")
```

B. Extensions for NodeSeq

```scala
val XQueryEquals = new Equiv[NodeSeq] {
  def equiv(a: NodeSeq, b: NodeSeq): Boolean = {
    a.map(_.text).toSet & b.map(_.text).toSet).size > 0
  }
}

implicit class nsExtensions(nodeSeq: NodeSeq) {
  def ===(b: NodeSeq): Boolean =
    XQueryEquals.equiv(nodeSeq, b)
  def ===(b: GenTraversable[String]): Boolean =
    b.exists(text =>
      XQueryEquals.equiv(nodeSeq, Text(text)))
  def ===(hasValue: String): Boolean =
    nodeSeq.text == hasValue
  def sortedByText: NodeSeq =
    nodeSeq.sortBy(_.text)
  def sortedText: Seq[String] =
    nodeSeq.map(_.text).sorted
  def sortedByLookup(ofPath: String): NodeSeq =
    nodeSeq.sortBy(node => (node \ ofPath).text)
  def groupByText(ofPath: String) :
    Map[String, NodeSeq] =
    nodeSeq.groupBy(node => (node \ ofPath).text)
  def groupByOrderBy(ofPath: String) :
    list[([String, NodeSeq])] =
    groupByText(ofPath).toList.sortBy(_._1)
  def textSet: Set[String] =
    nodeSeq.map(_.text).toSet
}
```