The key to good performing Schematron

Roger L. Costello
July 8, 2014

Recall that in Schematron documents rules are expressed using one or more <assert> elements, each using an XPath expression:

```
<sch:assert test="XPath expression">
   …
</sch:assert>
```

The way the XPath expressions are designed can make a huge difference in performance. For example, recently I was dealing with an XML document containing over 100,000 records. I needed Schematron to test each record. In my first version, executing the Schematron on the XML document took 3 hours. I then revised the XPath expressions and the time dropped to 6 seconds!

The example I present below isn't my actual case, but it's analogous enough to illustrate the idea.

Example

I have a list of <Book> elements. Each Book contains Title, Author, and Genre. Plus each Book has an id attribute.

Here is the rule to be enforced:

> For each Book there must not be another Book
> with the same Author and Genre.

This is the Schematron rule that I initially created:

```
<sch:rule context="Book">

   <sch:let name="currentAuthor" value="Author"/>
   <sch:let name="currentGenre" value="Genre"/>

   <sch:assert test="not(following-sibling::Book[Author eq $currentAuthor][Genre eq $currentGenre][1])">
      …
   </sch:assert>
</sch:rule>
```

Notice the XPath expression:

```
following-sibling::Book[Author eq $currentAuthor][Genre eq $currentGenre][1]
```

Evaluating that XPath is horribly expensive. To see why, suppose there are 100,000 Books. When evaluating the first Book, the XPath expression may (in the worst case) require examination of the following 99,999 Books. When evaluating the second Book, the XPath expression may (in the worst case) require examination of the following 99,998 Books. When evaluating the third Book, the XPath

expression may (in the worst case) require examination of the following 99,997 Books. And so forth. Thus, that XPath expression takes on the order of n^2 time to evaluate (n = the number of Books).

Yikes!

That Schematron program will take a very long time to execute!

The solution to better performance is to use XSLT keys. Yes, you can use XSLT keys in your Schematron. From Appendix C of ISO Schematron:

> The XSLT key element may be used, in the
> XSLT namespace, before the pattern elements.

At the top of the Schematron document add this:

```
<xsl:key name="books" match="Book" use="Author"/>
```

Then use the key() function in the XPath expression:

```
key('books', $currentAuthor)[@id ne $currentId][ Genre eq $currentGenre][1]
```

The <key> element instructs Schematron to index each Book in the XML document, based on Author. That makes querying Books by Author super-fast. That XPath executes very quickly.

Lesson Learned

The key to good performing Schematron is the XSLT <key> element coupled with the key() function.

Complete Schematron Document

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            queryBinding="xslt2" >

  <xsl:key name="books" match="Book" use="Author"/>

  <sch:pattern id="books">
    <sch:rule context="Book">

      <sch:let name="currentAuthor" value="Author"/>
      <sch:let name="currentGenre" value="Genre"/>
      <sch:let name="currentId" value="@id"/>

      <sch:assert test="not(key('books', $currentAuthor)[@id ne $currentId][ Genre eq $currentGenre][1])">

        A Book is unique in Author and Genre

      </sch:assert>
    </sch:rule>
  </sch:pattern>
```

```
</sch:schema>
```

Sample XML Document

```
<Books>
   <Book id="1">
      <Title>ABC</Title>
      <Author>John Doe</Author>
      <Genre>Fiction</Genre>
   </Book>
   <Book id="2">
      <Title>XYZ</Title>
      <Author>John Doe</Author>
      <Genre>Fiction</Genre>
   </Book>
   <Book id="3">
      <Title>Blah</Title>
      <Author>Sally Smith</Author>
      <Genre>Non-Fiction</Genre>
   </Book>

   …
</Books>
```


Acknowledgements

Thanks to the following people for their inputs:

- Syd Bauman
- David Birnbaum
- Michael Kay
- Wolfgang Laun
- Dimitre Novatchev
- Liam Quin
- Andrew Welch