Clean XML duplicate nodes using XSLT

19.10.23

Pegasi Knowledge
https://ghost.pegasi.fi/wiki/

Table of Contents

Clean XML duplicate nodes using XSLT

I am wrestling with SAP XML files. Those of you who know SAP XML dumps know that they are one big beasts with gigabytes of complex XML, separate node structures with no ids to bind them together and interesting German namings.

I am using XSLT to make them NetIQ IDM compatible XML but in my case there are lots of duplicate nodes and I want to get rid of them. I also need to do it efficiently without eating a terabyte of RAM so I have to do it with XSLT in a special way.

The source data

The XML output by SAP contains numerous parallel and serial paths to but no point in trying to demonstrate it here. For documentation's sake here is a snipplet of how it looks like as it enters the NetIQ IDM:

```
<ZHRMD_IAM>
  <IDOC BEGIN="1">
    <EDI_DC40 SEGMENT="1">
      <TABNAM>EDI_DC40</TABNAM>
      <MANDT>300</MANDT>
      <DOCNUM>0000000123456</DOCNUM>
      <DOCREL>123</DOCREL>
      ...continues....
```

Using NetIQ XSLT policy I've flattened the XPATH structures to XML attribute names consisting of infotype and element names concatenated with ':'. So now we have an IDM compatible XML file that looks something like this :

```
<nds dtdversion="1.1" ndsversion="8.6" xml:space="default">
  <input>
    <add class-name="Class" src-dn="1234">
      <association>1234</association>
      <add-attr attr-name="0001:PERNR">
        <value type="string">1234</value>
      </add-attr>
      <add-attr attr-name="0001:SEQNR">
        <value type="string">000</value>
      </add-attr>
      <add-attr attr-name="0001:INFTY">
        <value type="string">0001</value>
      </add-attr>
      <add-attr attr-name="0001:ENDDA">
```

```
        <value type="string">20181212</value>
      </add-attr>
      ..cut from here..
    </add>
  </input>
</nds>
```

Now I've got something to work with but I've also got a lot of duplicate nodesets that have no informational value. They only consume log space making my debugging harder.

Muenchian method

The first thing I tried is to use Muenchian method to clean out duplicate nodes. Muenchian method is a way to get unique valued nodeset from given nodeset. It utilizes keys and XSLT techniques to reach the conclusion with a minimal footprint. At least so they say.

The downside of this method in this case is the key functionality. While it is a fast method the keys it uses can only be initialized globally above the template match context therefore needing to index every single node from every single entry of your iDoc making it extremely slow with large files. See the traditional method from below for larger files.

Using Muenchian method requires us to list unique association-attribute-value combinations to reach every node and value of every person so we define a key accordingly :

```
<xsl:key name="attr-by-value" match="add-attr"
use="concat(../association,'+',@attr-name,'+',value)"/>
```

After which we can remove duplicates with add (E1PLOGI) matching, which seems to be fastest option of Muenchian method :

```
<xsl:template match="add[@class-name='person']">
    <add class-name="{@class-name}" src-dn="{@src-dn}">
        <association><xsl:value-of select="association"/></association>
        <xsl:for-each select="add-attr[count(. | key('attr-by-value',
concat($association,'+',@attr-name,'+',value))[1]) = 1]">
            <xsl:copy-of select="."/>
        </xsl:for-each>
    </add>
</xsl:template>
```

Also you can also use generate-id() to select unique nodes. But let it be noted that the performance is the same with it:

```
    <xsl:for-each select="add-attr[generate-id() = generate-id( key('attr-
by-value', concat(@attr-name,'+',value))[1] )]">
```

Or straight up you can do add-attr template matching, which is a compact piece of code but notably slower and working with smaller dumps :

```
    <xsl:template match="add-attr[not(generate-id() = generate-id(key('attr-by-value', concat(../association,'+',@attr-name,'+',value))[1]))]"/>
```

The aboce xpath selections generate a ton of iterations for a larger file so be aware of that.

The traditional method

This is the traditional way of doing the deduplication of nodes and it seems a better alternative for larger files.

We template match one E1PLOGI (=add) at a time and for-each select with attribute and value checks. Here is the code :

```
<xsl:template match="add[@class-name='element']">
    <add class-name="{@class-name}" src-dn="{@src-dn}">
        <association><xsl:value-of select="association"/></association>
        <xsl:for-each select="add-attr[@attr-name!=preceding-sibling::add-attr[value=preceding-sibling::add-attr/value]/@attr-name]">
            <xsl:copy-of select="."/>
        </xsl:for-each>
    </add>
</xsl:template>
```

The optimized method

The methods above are all inefficient due to multiple concurrent loops. Whether you use for-each or direct template matching something like following will happen:

- for loop goes through all of the nodes
- direct template match goes through all of the nodes
- key functionality will query all attributes and create nodesets from all of them with every single attribute in a loop and compare them
- the last method will query all attribute names from all attributes with same values and compares both

A single person entry in SAP XML is hundreds or even thousands of rows - resulting to a mess.

An optimized and fully working way is to sort the nodes with attribute name and value, compare against following nodes (vs all of the nodes) and copy / skip based on that. It is a big improvement to the above. Here is the code :

```
<xsl:template match="add[@class-name='element']">
    <add class-name="{@class-name}" src-dn="{@src-dn}">
        <association><xsl:value-of select="association"/></association>
        <xsl:for-each select="add-attr">
            <xsl:sort select="@attr-name"/>
            <xsl:sort select="value"/>
            <xsl:if test="not(@attr-name = following-sibling::add-
attr/@attr-name) or not(value = following-sibling::add-attr/value)">
                <xsl:copy-of select="."/>
            </xsl:if>
        </xsl:for-each>
    </add>
</xsl:template>
```

The very optimized but compromised method

The most time optimized way is to sort the nodes with attribute name and value, compare only against next node (vs all of the nodes) and copy / skip based on that. This is a huge the winner when performance is concerned but the result is not 100% guaranteed. See the results at the end of this page. Here is the code :

```
<xsl:template match="add[@class-name='element']">
    <add class-name="{@class-name}" src-dn="{@src-dn}">
        <association><xsl:value-of select="association"/></association>
        <xsl:for-each select="add-attr">
            <xsl:sort select="@attr-name"/>
            <xsl:sort select="value"/>
            <xsl:if test="not(@attr-name = following-sibling::add-
attr[1]/@attr-name) or not(value = following-sibling::add-attr[1]/value)">
                <xsl:copy-of select="."/>
            </xsl:if>
        </xsl:for-each>
    </add>
</xsl:template>
```

Why is it not guaranteed working? Even though the source material is sorted there is no guarantee that following-sibling returns the node set in the same order. You can improve the odds by sorting the material in previous policy but still it seems not to be 100% accurate with large material. So we must give up the next node checking and settle with checking the rest of the nodes - unless you are happy with some occasional duplicates showing up.

Heavily optimized AND working method

I decided to try a very straightforward way to first sort my XML in it's own policy and use counter with compare to the next element. This resulted in execution time reduction to roughly 1/5 of the time it took to do the optimized version! That is fast! Here is the code :

```
<xsl:template match="add[@class-name='element']">
    <add class-name="{@class-name}" src-dn="{@src-dn}">
        <association><xsl:value-of select="association"/></association>
        <xsl:for-each select="add-attr">
            <xsl:variable name="position" select="position()"/>
            <xsl:if test="not(@attr-name = ../add-
attr[position()=($position+1)]/@attr-name) or not(value = ../add-
attr[position()=($position+1)]/value)">
                <xsl:copy-of select="."/>
            </xsl:if>
        </xsl:for-each>
    </add>
</xsl:template>
```

Keys or no keys

At first I tried to reuse same keys per person such as :

```
<xsl:key name="attr-by-name" match="add-attr" use="@attr-name"/>
```

But it resulted in illogical outcomes indicating that a key entry must be unique per stylesheet / XML file, as in the Muenchian example earlier in this document. You cannot have a key with attr-name and use it over again per user in the same XML document. The key index will be allocated and by trying to reallocate it you will end up with unexpected results.

If you have multiple repeating entries with similar attributes you need to create a unique key entry per XML document and that may be a showstopper. In my case it would have resulted in hundreds of thousands indexes. So for larger XML files keys are a no-go.

Sort or no sort

Sorting is the key to solution here. Sorting in XSLT is quick. Very quick. And now when we're sorted we can compare the current node only against the next one! That is a huge win.

Results and conclusion

My test material was around 7000 rows of XML containing roughy 80 users' data. Here are some times to illustrate the effectiveness of different methods :

IDM scripting language method

- Roughly 21 minutes
- Very simple to do
- Easily debuggable with IDM tracing

Muenchian method

- Using unique keys / person, template matching each person
- Roughly 6-7 minutes
- Did not produce correct results due to key index recycling
- Unusable

Muenchian method with globally indexed keys

- Using unique keys globally, template matcing each person
- Did not complete in 1 hour, did not wait longer
- Unusable

The optimized method

- Template matching each person
- Sorting the input
- Comparing the results to the remaining nodes
- 56 seconds

The very optimized method

- Template matching each person
- Comparing the results to the next node
- 3.8 seconds
- Not 100% accurate results

The heavily optimized AND working method

- Template matching each person
- Comparing the results to the next node using node positioning
- Roughly 10 seconds
- Works

Quick words

This document is a result of multiple tweaks, mistakes and tests. You can make a big difference on how fast your XML is processed and strangely enough you can even make a compromise between accuracy and performance. XSLT with two axis sorting cuts down the required number of operations as well as memory usage very dramatically but it seems that following-sibling statement is possibly so optimized and operating on memory's terms that is does not guarantee the order of things? Hope to get some second opinions on that.