

Handling XSL:FO's memory issue with large page counts

WHITE PAPER



David Kelly
Senior Technical
Consultant

Formatting Object (FO) processors (FOP, in particular) often fail with memory errors when processing very large documents for PDF output. Typically in XSL:FO, the body of a document is contained in a single fo:page-sequence element. When FO documents are converted to PDF output, the FO processor holds an entire fo:page-sequence in memory to perform pagination adjustments over the span of the sequence. Very large page counts can result in memory overflows or Java heap space errors. Reducing page count in a document is not usually an option.

You can set up processing to divide the document into multiple fo:page sequences to avoid memory problems. The granularity of the content of each fo:page-sequence does not matter. Each fo:page-sequence could contain a chapter, a sub-chapter, a section, and so on. For a DITA implementation, each DITA topic could be placed in its own page sequence. The key is to place elements in a flat sequence of fo:page-sequences.

Flattening the hierarchy is challenging. The examples in this document will explain the approach.

NOTE: This document assumes basic familiarity with XSL, XSL:FO, DITA, and the DITA Open Toolkit.

Considerations in creating multiple page sequences

One problem with modifying page sequences is that fo:page-sequences cannot contain other fo:page-sequences. The XSL that creates the FO output must chunk the content and organize the chunks as peers at the level where the chunks will be divided into fo:page-sequences.

If your chunks are chapters, they are probably already peers. Inside the chapters, the organization of elements can remain the same. The chapter chunks need to be wrapped in their own fo:page-sequence elements.



Another issue in creating multiple page sequences is that each `fo:page-sequence` forces the start of a new page. This is probably not an issue if you divide a book into one `fo:page-sequence` per chapter, but putting very short sections inside separate `fo:page-sequences` would lead to a lot of empty paper.

Creating a sequential set of output based on a hierarchically ordered set of inputs might suggest that there would be a change in formatting. Formatting issues are not a problem when applying this approach to the DITA Open Toolkit (OT). The division of page sequences does not affect numbering streams because those streams are dependent on the organization of the input document—not the output document. Cross-references are typically based on positions in the input document; placement in separate page sequences does not affect the resolution of `fo:internal-destination` elements in the output document.

If indentations for elements at different levels in the hierarchy are always relative to the page margin, the indentation is not dependent on the relationship of child blocks to parent blocks in the output. The relationship of these blocks can be altered without changing the physical indentation.

This kind of relationship turns out, in fact, to be the case for the DITA OT. No changes to indentations are required in the DITA OT for this solution. For other implementations, you may need to adjust formatting so indentation is not dependent on the hierarchical relationships of blocks in the output.

XML document structure and `fo:page-sequences`

In an XML document, you might have a structure similar to [Example 1](#).

Example 1

```
<book>
  <section>
    <body>...</body>
    <section>
      <body> </body>
    </section>
  </section>
</book>
```

Now imagine that the second-level sections contain hundreds of additional sections, many embedded in each other, and the document is 10,000 pages long.

The FO output for the body of the example document is similar to [Example 2](#):

Example 2

```
<fo:page-sequence>
  <fo:flow>
    <fo:block> <!-- book -->
      <fo:block> <!-- section -->
        <fo:block> <!-- section body-->
      </fo:block>
```

```

        <fo:block> <!-- child section -->
            <fo:block> <!-- child section body -->
            </fo:block>
        </fo:block>
    </fo:block>
</fo:flow>
</fo:page-sequence>

```

To break the document into multiple page sequences down to the second-level section, the output would resemble [Example 3](#):

Example 3

```

<fo:page-sequence>
  <fo:flow>
    <fo:block> <!-- section -->
      <fo:block> <!-- section body -->
      </fo:block>
    </fo:block>
  </fo:flow>
</fo:page-sequence>
<fo:page-sequence>
  <fo:flow>
    <fo:block> <!-- child section-->
      <fo:block> <!-- child section body -->
      </fo:block>
    </fo:block>
  </fo:flow>
</fo:page-sequence>

```

The block for the child section is at the same level as the block for the section, but it is in a separate page sequence. Formatting and numbering are based on the XML tree in the source document and not the output document, so they are preserved.

The FO output has been flattened into a sequence of `fo:page-sequence` elements. Applying this pattern to the 10,000 page document would give you hundreds or thousands of `fo:page-sequence` elements—and no Java memory problems.

The default XSL:FO approach

[“Example 4” on page 4](#) shows XSL templates that create a single page sequence. The output from this XSL is a set of hierarchically organized `fo:blocks` within the single page sequence. Notice the relationship of the `<fo:page-sequence>` element to the `<xsl:apply-templates>` element — one contained in the other.

These templates generate the output in [Example 2](#), in which a single `fo:page-sequence` and `fo:flow` are wrapped around all the top-level section elements.



Example 4

```

<xsl:template match="/book">
  <fo:page-sequence>
    <fo:flow>
      <fo:block>
        <xsl:apply-templates select="section"/>
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
</xsl:template>
<xsl:template match="section">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
<xsl:template match="body">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

```

The XSL:FO solution

To break the content into multiple page sequences, you modify the XSL templates to defer the creation of `fo:page-sequence` and `fo:flow` until it gets to the section element, specifically the top-level sections and the second-level sections. These functions are accomplished with two templates for sections with different modes. The solution uses a condition for selecting one template or the other based on the level of the section within the hierarchy.

The following code shows the solution. It generates the output in [“Example 3” on page 3](#) (the “good” example) from the input in [“Example 1” on page 2](#).

NOTE: The code shown is *conceptual code*. It is based on working and tested code, but has been simplified and has not itself been tested.

Example 5

```

<xsl:template match="/book">
  <xsl:apply-templates select="section" mode="page"/>
  <!-- This takes care of all the top-level section tags. -->
</xsl:template>
<!-- The following template matches section at the top level and second level. -->
<xsl:template match="section" mode="page">
  <fo:page-sequence>
    <fo:flow>
      <xsl:apply-templates select="body"/>
      <xsl:if test="count(ancestor::section)=2"/>
        <xsl:apply-templates select="section"/>
      </xsl:if>
    </fo:flow>
  </fo:page-sequence>
</xsl:template>

```

The preceding “if” condition selects third-level sections while it is processing a second-level section. You could adjust the condition to select any level of section you wanted, depending on how far down you wanted to start chunking. At this point, processing of section elements is taken over by the other “section” template. This way, the hierarchy of descendant sections is preserved within the current fo:page-sequence.

```
</fo:flow>
</fo:page-sequence>
<xsl:if test="count(ancestor::section) &lt; 2">
```

The preceding condition selects only the second-level sections. You could adjust this to any level of section you wanted, depending on how far down you wanted to start the chunking. (You could also create a “chunking” variable to set at the top of this template so you could change it in only one place instead of two.)

```
<xsl:apply-templates select="section" mode="page"/>
```

This apply-templates invokes the current template, so the second-level sections will have the page sequence wrapped around them as well. The fo:page-sequence tag in this template is closed *before* this apply-templates is invoked. This is how the element hierarchy is flattened into a series of fo:page-sequences.

```
</xsl:if>
</xsl:template>
<xsl:template match="section">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
<xsl:template match="body">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Two tricks worth mentioning are:

- ❖ Wrap the fo:page-sequence tags around the elements that need to be chunked but not around the child elements that don’t need to be chunked.
- ❖ Set up conditions to specify which elements to chunk and which not.

For the DITA OT, the solution is a little trickier. That solution is described in the following sections.

The default implementation in the DITA Open Toolkit: single page sequence

Two templates in two stylesheets handle the topic element in the DITA OT: xsl/dita2fo-shell.xsl and xsl/xslfo/topic2foimpl.xsl. Which template handles the topic element depends on whether you are using the DITA map element to organize your books.



[Example 6](#) shows the DITA-OT template in `dita-ot/xsl/dita2fo-shell.xsl` that creates a single `fo:page-sequence` wrapping around all the topics within the map. This template is called from the **dita-setup** template. The context of this template is the map element. (Actually, the context is defined by `match="[contains(@class,' map/map')]"`, to be accurate.)

This style sheet operates on the temporary, merged XML file for the book in the build directory in the DITA OT, not directly on the source ditamap. In the temporary merged XML file, the map element surrounds all the topic content, not the topicrefs that it surrounds in the ditamap files. So even though the context is the map element, when this template issues the `<xsl:apply-templates>` element, the element it works on is a topic (or an element related to a topic).

Notice, as you did in [Example 4](#), the relationship between the `xsl:apply-template` element for topics and the `fo:page-sequence` element. One is contained inside the other.

Example 6

```
<xsl:template name=" main-doc3">
  <fo:page-sequence master-reference="chapter-master">
    <fo:static-content flow-name="xsl-region-before">
      ...
    </fo:static-content>
    <fo:static-content flow-name="xsl-region-after">
      ...
    </fo:static content>
    <fo:flow flow-name="xsl-region-body">
      <fo:block text-align="left" font-size="10pt" font-family="Helvetica" break-before="page">
        <xsl:apply-templates/>
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
</xsl:template>
```

[Example 7](#) shows the DITA-OT template in `xsl/xslfo/topic2foImpl.xsl` that matches a top-level topic when there is no ditamap:

Example 7

```
<xsl:template match="*[contains(@class,' topic/topic ')]" name="toptopic" mode="toplevel">
  <xsl:call-template name="chapter-setup"/>
</xsl:template>
```

The following template matches all other topics, whether they are embedded in a map or a top-level topic. This template only creates `fo:block` elements, not `fo:page-sequence` elements.

```
<xsl:template match="*contains(@class,' topic/topic ')">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The following template provides the `fo:root` and `fo:page-sequence` elements when the top-

level topics are not contained in a map element. Just like the structure in [Example 6](#), the `xsl:apply-template` element for lower-level topics is contained inside the `fo:page-sequence` element.

```
<xsl:template name="chapter-setup">
  <!--Newline character (capture the native file newline)-->
  <xsl:variable name="newline"/>
  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
      ...
    </fo:layout-master set>
    <fo:page-sequence master-reference="common-page" force-page-count="no-force">
      <fo:static-content flow-name="xsl-region-before" font-size="9pt" font-family="Helvetica">
        ...
      </fo:static-content>
      <fo:static-content flow-name="xsl-region-after">
        ...
      </fo:static-content>
    <!-- body -->
    <fo:flow flow-name="xsl-region-body">
      <fo:block line-height="10pt" font-size="9pt" font-family="Helvetica" id="page1-1">
        <xsl:apply-templates/>
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
</xsl:template>
```

It is important to remember that in the DITA-OT, DITA files can be processed in two ways: as stand-alone DITA documents or as ditamaps. The `dita2fo-shell.xsl` file sets up the page flow when a ditamap is being processed. If a single DITA file is processed, the document is structured with the top element having a `class` attribute that contains the string `topic/topic`. The top-level topic is processed with the template:

```
<xsl:template match="*[contains(@class,'topic/topic')]" name="toptopic" mode="toplevel">
```

which in turn calls the template:

```
<xsl:template name="chapter-setup">
```

For both processing as a stand-alone DITA document or a ditamap, topics beneath the map element or the top-level topic are processed by the template that begins:

```
<xsl:template match="*contains(@class,'topic/topic')">
```

But by this time, the `fo:page-sequence` has already been created, so the whole book is now in a single large `fo:page-sequence`. This is the root cause of the memory problem for large books.



The solution in the DITA Open Toolkit: multiple page sequences

Modifying the transformation process to create multiple page sequences and preventing them from wrapping each other eliminates the memory problem. [Example 8](#) and [Example 9](#) show the solution. The altered code processes all topics and subtopics inside their own page sequences. Therefore, the map template does not start any page sequence at all. Each topic template (including the top-level topic template) creates a page sequence and also places the <apply-templates> for child topics *outside* the fo:page-sequence element, so fo:page-sequences are not nested.

[Example 8](#) shows the changes in dita-ot/xsl/dita2fo-shell.xsl.

NOTE: Bold type indicates where code is added to or deleted from the original templates.

Example 8

```
<xsl:template name=" main-doc3 ">
  <!--deleted all the page sequence stuff. -->
  <xsl:apply-templates></xsl:apply-templates>
</xsl:template>
```

[Example 9](#) shows the modified templates in dita-ot/xsl/topic2foimpl.xsl.

Example 9

```
<xsl:template match=" *[contains(@contains(@class,' topic/topic '))" name="toptopic" mode="toplevel">
  <xsl:call-template name=" chapter-setup "/>
</xsl:template>
<xsl:template match=" *[contains(@class,' topic/topic ')]">
  <fo:page-sequence master-reference="chapter-master">
  ...
  <fo:static-content flow-name="xsl-region-before">
  </fo:static-content>
  <fo:static-content flow-name="xsl-region-after">
  </fo:static-content>
  <fo:flow flow-name="xsl-region-body">
```

Now, instead of a generic apply-templates, use select attributes to process child elements in the same order they occur in DITA. However, place the apply-templates that selects child topics outside the fo:page-sequence tags.

```
      <fo:block>
<!-- The generic apply-templates is replaced with specific
apply-templates -->
        <xsl:apply-templates select="*[contains(@class,' topic/title ')]"/>
        <xsl:apply-templates select="*[contains(@class,' topic/titlealts ')]"/>
        <xsl:apply-templates select="*[contains(@class,' topic/shortdesc ')]"/>
        <xsl:apply-templates select="*[contains(@class,' topic/prolog ')]"/>
        <xsl:apply-templates select="*[contains(@class,' topic/body ')]"/>
        <xsl:apply-templates select="*[contains(@class,' topic/related-links ')]"/>
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
```

```

<!-- The apply-templates for child topics is invoked outside the page sequence. -->
  <xsl:apply-templates select="*[contains(@class,' topic/topic ')]"/>
</xsl:template>
<xsl:template name="chapter-setup">
<!--Newline character (capture the native file newline)-->
  <xsl:variable name="newline"/>
  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
      ...
    </fo:layout-master set>
    <fo:page-sequence master-reference="common-page" force-page-count="no-force">
      <fo:static-content flow-name="xsl-region-before" font-size="9pt" font-family="Helvetica">
        ...
      </fo:static-content>
      <fo:static-content flow-name="xsl-region-after">
        ...
      </fo:static-content>
    <!-- body -->
      <fo:flow flow-name="xsl-region-body">

```

Again, instead of a generic `apply-templates`, use `select` attributes to process child elements in the same order they occur in DITA. Place the `apply-templates` that select children topics outside the page sequence tags.

```

      <xsl:apply-templates select="*[contains(@class,' topic/title ')]"/>
      <xsl:apply-templates select="*[contains(@class,' topic/titlealts ')]"/>
      <xsl:apply-templates select="*[contains(@class,' topic/shortdesc ')]"/>
      <xsl:apply-templates select="*[contains(@class,' topic/prolog ')]"/>
      <xsl:apply-templates select="*[contains(@class,' topic/body ')]"/>
      <xsl:apply-templates select="*[contains(@class,' topic/related-links ')]"/>
    </fo:flow>
  </fo:page-sequence>
  <xsl:apply-templates select="*[contains(@class,' topic/topic ')]"/>
</fo:root>
</xsl:template>

```

This solution puts every topic in its own page sequence. This approach is adjustable and can separate topics into individual page sequences down to a specified level in the hierarchy, as described in [The default XSL:FO approach](#).

Conclusion

XSL:FO is a great way to create PDF files from XML, but very large PDF files present memory issues. The cause is frequently the use of a single `fo:page-sequence` to contain the entire body of a book. Chunking the book into multiple `fo:page-sequences` can prevent these memory problems. The XSL:FO needs to generate page sequences with each child element parallel to the parent's page sequences—not embedded in them.



This approach is especially useful for DITA because it embeds the same XML element (“topic”) at many levels. The strategy outlined in this document has the flexibility to chunk topics down to a specified level. Using this approach, forced page breaks are kept to a minimum while fo:page-sequence sections are adjusted for the memory limitations of the FO processor.

About the author

David Kelly has toiled in the fields of technical communications since 1978 and has filled a variety of roles: technical writer, publications coordinator, documentation manager, project manager, and technical consultant, just to name a few. Now, at the pinnacle of his career, he hacks at the DITA Open Toolkit for pleasure and profit.

In addition to his many professional accomplishments, David has published poetry and two novels, won awards for his photography, once designed and built a sailboat, and currently creates large, colorful oil paintings, some of which decorate Scriptorium’s office.

About Scriptorium

Scriptorium Publishing provides expert advice on how to develop, deploy, and manage content. Our typical customer has thousands of pages of information, which needs to be delivered in print, PDF, HTML, and other media, often in dozens of languages. Our mission is to automate formatting and production tasks, usually through XML technologies, so that authors can write more efficiently.

Our consultants have experience in traditional publishing workflows, including typesetting, book design, copyfitting, and production editing. This understanding influences our approach to creating state-of-the-art publishing systems with modern tools and technologies, such as XML, HTML, DITA, the DITA Open Toolkit, XSLT, XSL-FO, FrameMaker, Ant, Perl, FrameScript, Flash, InDesign, XMetaL, oXygen, and many more.

Our customers include federal and state government as well as companies in defense, consumer electronics, telecommunications, health care, pharmaceutical, and other industries.

If you are facing a difficult publishing challenge, we want to hear from you. Contact us after the conference at info@scriptorium.com or 919-481-2701 x105.

Scriptorium Publishing is based in the Research Triangle area of North Carolina and has been in business since 1997.

Scriptorium Publishing Services, Inc.
PO Box 12761
Research Triangle Park, NC 27709-2761
USA
info@scriptorium.com
919-481-2701
www.scriptorium.com

