Abstract

An XML transformation, commonly specified in XSLT, is an important step during XML-based data migration. Since common XSLT processors always parse input data into DOM-like structure, processing of large data sets to be migrated represents a significant problem; all available memory can be exhausted, the transformation takes unacceptable long time, or even the processors can fail. In this survey we explore current state-of-the-art in XML transformation processing on large data sets and present a performance comparison of DOM-based XSLT processors as well as streaming XSLT and STX processors. We show that there is no silver bullet among XSLT processing on large data sets; particular approaches have their strengths and limitations.

1 Introduction

Large data set migration is a common issue to be solved in enterprise data management. Certain data need to be transferred between different storages and schemas in order to be accessible for different systems. XML is often chosen as a temporary representation of migrated data due to its simplicity, portability and number of supporting technologies. Moreover, XML is becoming a standard export/import format in many enterprise applications as well as common DBMS. Such XML-based migration proceeds in three steps:

- Exporting data into XML format in the source system (data extraction).
- Transforming data.
- Importing data from XML format in target system (data loading).

Step 1, step 3 or both of them are skipped in case the XML format itself is used for data storage in the source system, in the target system or in both of them, respectively. The XML transformation is almost inevitable for two reasons:

1. The extracted data still need some restructuring in order to adhere to the target schema.
2. The export XML schema of the source system differs from the import XML schema of the target system - the differences may be textual (different tag names, different attribute names, etc.) as well as structural (different hierarchy and placement of metadata information - ordering, child-parent relationship, using element vs. attribute, etc.).

Since each logical unit (table) of migrated data consists of records, the XML representation contains large number of structurally equal patterns. The modifications needed typically apply on each such pattern separately and thus the changes required is local with respect to the pattern. Most of the global modifications can be captured during the data extraction phase.

This paper explores current state-of-the-art in XML transformation processing on data sets of various sizes. We model the conditions specific for data migration - the input data are generated by repeating certain pattern within the XML document, and the sample transformation specifies local changes with respect to such pattern and one simple global transposition. According to our best knowledge, no XSLT benchmarking on large data sets has been presented until now - see also recent XML-related benchmark survey [19]. The XSLTMark benchmark [17] covered only small datasets and is out-of-date today. We hence try to fill a gap in knowledge and provide an effective evaluation of what current XSLT processors offer.

We focus on transformations written in high-level transformation languages supported by automatic processors. This approach brings several advantages comparing to programming transformation manually using a SAX-based parser:
• No time-consuming low-level programming is needed.

• The migrator is only supposed to be familiar with the high-level specification language, no SAX programming skills are required.

• The high-level transformation specification is clearer and easier to maintain in comparison to hard-coded transformation.

W3C XSLT language is a natural choice for specifying XML transformations. It is supported by a number of processors. Leading XSLT processors are DOM-based, i.e., they store the whole input data in the memory and then perform the transformation according to the specification. Although much effort has been devoted to make the processors more efficient, primarily by optimizing data structures used for in-memory storage, the memory usage still remains proportional to the size of the input data. The streaming processing is an alternative to the DOM-based processing. In optimal case, the processor stores as much of the input data in the memory as needed at a given moment. This approach is algorithmically much more difficult than the DOM-based processing since it is necessary to identify parts of the input data to be buffered temporarily. We include four DOM-based XSLT processors and one prototype streaming XSLT processor in our tests.

Other transformation languages (STX [4], CDuce [6], XStream [12]) are rather experimental and used rarely in enterprise environments. However, we complement our tests with one STX processor since STX is a promising representative of streaming specification languages - an interesting trade-off between low-level SAX programming and high-level XSLT.

The paper is organized as follows: Section 2 contains brief description of the processors evaluated. In Section 3, we specify settings for our tests and present the results of the performance measurements. In Section 4, the behavior of particular processors during the tests is described in more detail. Finally in Section 5 we interpret the results obtained, compare them to our expectations and mention proposals for future development of XSLT processors in order to get efficient solution for large input data sets.

2 Evaluated Processors

We present results of experiments performed on several common XML transformation processors. We have tested four DOM-based XSLT processors:

**Saxon-SA** [16] is an XSLT processor written in Java, an enhanced version of the open source Saxon-B processor. It provides a possibility to process parts of the input document sequentially, however, these parts need to be selected manually in XSLT stylesheet using special directives.

**AltovaXML** [1] is a set of XML utilities which includes an XSLT processor used in Altova XML tools XMLSpy, MapForce, and StyleVision.

**Xalan** [3] is an open source software library from the Apache Software Foundation, originally created by IBM under the name LotusXSL that implements the XSLT transformation language and the XPath language. The Xalan XSLT processor is available for both the Java and C++ programming languages.

**XsltProc** [20] is a fast command-line XSLT engine. It is a part of the GNOME XSLT library.

We have included two streaming processors of XML transformations - the first one processes W3C XSLT stylesheets while the second one uses its own specification language.

**Xord** [11] is an engine for streaming, memory-efficient processing of XSLT transformation. It contains various analyzing and transforming algorithms that differ in their power - each of them is able process different subset of XSLT. In this paper we use the BUXT module of this engine since it captures the largest XSLT subset. The engine is based on a set of formal models called streaming XML transducers [8], each covering different class of XSLT stylesheets.

**Joost** [5] is a Java processor of STX language that is intended to describe streaming XML transformations specifically. STX syntax reminds XSLT syntax, the important feature is addition of special directives for input/output/temporary buffering. We have not tested the Perl STX processor XML::STX [13] since its development terminated in 2003 and it is currently not available for public download.

3 Performance Measurement

All the measurements were accomplished using the same operating environment: Intel Core 2 Duo processor, 2 GB RAM, Windows XP 32-bit operating system. No other applications or additional services were running besides the processor currently measured. Fig. 1 shows the XML schema used in a regular-expression notation [10]. The basic sample XML file contains all elements specified in the schema exactly once, i.e., it is of the form  
\[a/(i1/12/b/(i6/c),c),(i3/14/15/b/(i6/c),c)\]  
using the flat notation. The larger files were obtained by repeating the pattern  
\[(i3/14/15/b/(i6/c),c)\]  
within the root element **a**.
a/(i1/i2/b/(i6/c),c),(i3+/i4/i5/b/(i6/c),c)

**Figure 1. Sample XML schema**

01: `<xsl:template match="a" mode="m0">
02:     <output_a1>
03:         <xsl:apply-templates
04:             select="i3/i4/i5/b" mode="m2"/>
05:         <xsl:apply-templates
06:             select="i1/i2/b" mode="m1"/>
07:     </output_a1>
08: </xsl:template>
09: <xsl:template match="b" mode="m1">
10:     <output_b>
11:         <xsl:apply-templates
12:             select="i6/c" mode="m2"/>
13:         <xsl:apply-templates
14:             select="c" mode="m2"/>
15:     </output_b>
16: </xsl:template>
17: <xsl:template match="b" mode="m2">
18:     <output_b1>
19:         <output_b2/>
20:     </output_b1>
21: </xsl:template>
22: <xsl:template match="c" mode="m2">
23:     <output_c/>
24: </xsl:template>

**Figure 2. Sample XSLT stylesheet**

The sample XSLT stylesheet used is listed in Fig. 2. Since the evaluation is focused on processing large XML data during a data migration, we used simple transformation that conforms to the typical transformations on large data - selections, projections and a simple global transposition with respect to the input schema (rows 3,4). Note that the element b selected in row 3 is always positioned after the element b selected in row 4 in the sample input data and thus the processing order of these elements differ from their positioning in the input document.

More complex operations such as looping and sorting were intentionally not considered. This mode of processing enables to compare different processing techniques using equal conditions, namely DOM-based XSLT processing, streaming XSLT processing and STX processing.

Figure 3 contains numerical results of the measurements. Each processor was transforming a sample input data starting at 1 million of tags (1 Mt) until it crashes or the processing time exceeds a reasonable limit. At each run the elapsed time (in seconds - s) and the peak memory consumption (in megabytes - MB) were recorded. Time '99999' means that the processor crashed.

Fig. 4 to Fig. 6 depict these values in much more evident way. The horizontal axes represent size of data (in Mt - millions of tags), the vertical axes represent the amount of allocated memory (MB) or elapsed time (s). The time graph is zoomed to an area of intersections in Fig. 6 so that the behavior on smaller data sets (when the processing is still correct) can be seen in more detail.

4 Evaluation

4.1 DOM-based XSLT Processors

The processors Saxon-SA, AltovaXML and XsltProc behave almost equally on large data sets - the processing is first very efficient (only few seconds) until all available memory is consumed. Then the elapsed time grows significantly and the processors crash. From among these processors, XsltProc is the most efficient one, but the differences are not significant. There are notable differences in maximal processed data size - AltovaXML ends at 4 Mt, XsltProc at 16 Mt (with significant performance degradation over 14 Mt) while Saxon is able to process up to 18 Mt.
The Xalan processor behaves differently - its performance is very low even at few Mt, its time complexity is worse than linear. Processing of 18 Mt (that can be processed by Saxon within 1 minute) takes more than 2.5 hours. Nevertheless, unlike the other processors, it consumes notably less memory. Hence it is able to process much larger data sets at the expense of time. The largest data set, 30 Mt, was processed after more than 7 hours.

The results imply the following suggestion based on the input data size:

- \( \leq 16 \text{ Mt} \) - XsltProc (preferably) or Saxon,
- 16 to 18 Mt - Saxon,
- \( \geq 18 \text{ Mt} \) - Xalan - in case time is not important.

### 4.2 Streaming XSLT Processing

The main motivation behind the research on the streaming XSLT processing is limitation of data size that can be processed by the DOM-based processors. Although there have been published some papers about formal streaming models [7, 14, 15], the only known implementation is the Xord project [9].

Our measurements show that the memory complexity of the streaming transformation does not depend on the input size what exactly conforms to the behavior of the underlying theoretical models - streaming XML transducers [8]. On the other hand, especially Fig. 6 makes evident that the performance of the streaming processors is significantly worse than DOM-based processors. It is caused mainly by two reasons:

- the Xord framework is targeted to experiments and research of streaming algorithms, there are no performance optimizations implemented,
- since all input data must be processed sequentially, many computations (e.g., XPath evaluation) are performed redundantly.

Moreover, streaming processing does not provide full
XSLT functionality, only small subset is currently covered (no variables, no predicates, only apply-templates construction, only forward references etc.) Therefore, using streaming processing on data that can be processed by any DOM-based processor is meaningless. But, many currently available XML data sources are much bigger, e.g., well-known DBLP 'database' [18] contains about 700 MB of data, this is out of range of all DOM-based processors. In that case, even such restricted capabilities of streaming processing can provide required functionality, especially selection, projection and some transposition.

4.3 Other Processing Techniques

There are other techniques for processing large XML data. One of the most promising is STX [4]. We have rewritten the sample XSLT stylesheet to the STX language. The first problem is the complexity and cumbersomeness of the language. The simple and straightforward XSLT transformation shown in Fig. 2 consists of 140 lines in STX and looks very complicated, user must care about a lot of low-level technical details, explicitly declare, fill and use buffers etc. The Joost implementation of STX was not able to process even 1 Mt data - the program reported uncaught Java exception after 80 MB of allocated memory. Therefore we did not include this processor in the graphs. It seems that, despite promising idea, the current STX processors are not usable for large data.

Finally, the transformation can be written in any common programming language like Java, C++ or C#. Using any SAX implementation, the processing can be programmed in the sequential manner. However, all the technical details of the transformation must be programmed manually, the resulting code is very dependent on particular programmer skills and the development of such transformation is much more complex than high-level design of XSLT stylesheet.

5 Conclusions

This paper shows that there is no silver bullet in large XML data processing. A lot of conditions can influence the results that were measured and presented:

- **Physical memory size**: For DOM-based XSLT processors, this is the most relevant factor with major impact since it directly implies how large data sets can be processed.
- **Processor and OS architecture**: Comparing to the 32-bit environments, the 64-bit environments consume slightly more memory but at the same time they allow the processes to use larger amount of memory (no 4 GB limit).
- **Operating system**: Some XSLT processors are optimized for one class of OS.
- **Raw document size**: Especially the memory consumption of DOM-based processors is affected by the raw input data size such as the length of the tag and attribute names, the amount of textual content within elements and attributes.
- **Document depth**: The BUXT processor always allocates at least memory of the size proportional to the depth of the input XML document.
- **Input schema and XSLT stylesheet**: The BUXT processor is able to process only XSLT stylesheets that does not imply global transformations with respect to the input schema. The DOM-based processors are able to process any transformation, however, in case of complex modifications the time consumption may grow significantly. The STX processors are able to process any transformation as well, but according to our experience it is very complicated to rewrite a complex XSLT stylesheet (with large number of templates, loops, predicates) into equivalent STX stylesheet.

... and many others.

Although the performance presented in this paper has been measured using one pair (input schema, XSLT stylesheet) and a uniform operating environment, our previous experiments [11] using different classes of XSD/XSLT performed under different settings support the assumption that the results are valid generally. For different settings, the corresponding results differ only in the input data size for which the respective processing technique fails; the overall characteristics always remains equal.

Based on our previous experience, some of the results were expected. Nevertheless, other observations were more interesting.

**Expected results:**

- The DOM-based processors are significantly more efficient than the streaming processors until the memory is exhausted.
- Large data that does not fit into the memory must be processed by some streaming technique.
Noteworthy results:

- Almost no DOM-based XSLT processor is stable - besides Xalan every processor crashed after exhausting all available memory. Xalan processing on large data was too slow to reach the memory limit. The reported unstability is caused purely by the data size, it is not influenced by other aspects like the schema or the transformations used.
- There are significant differences in effective limitations of particular DOM-based processors.
- The limit of DOM-based processors is quite low, much lower than we expected. In our settings the maximal size of the processed data was about 250 MB (after more than 7 hours!). In reasonable time (within 1 minute), the maximal data size processed was about 180 MB
- The STX language, despite its promising idea, is not currently useful at all for large data since its implementations are not stable.
- Even if some STX implementation is stable, its code is much more complex than the corresponding XSLT code for transformations that require buffering.

The results show that there is a lot of issues in large XML data processing. The DOM-based transformers should improve their stability in case of shortage of available memory. Nevertheless, the nature of DOM model never allows the transformers to process data sets of the size exceeding certain limit.

Currently, the streaming XSLT processing is not practically applicable mainly due to very restrictive subset of XSLT captured. The following research should be focused on enhancing its features in order to allow processing of more advanced constructs. Moreover, the BUXT streaming processor suffers from complex design since the low-level buffering as well as the processing of transformation itself are contained within the same module.

According to our opinion, a trade-off between the two techniques will lead to and efficient and easily maintainable solution. Perhaps some emerging methods of parsing and processing XML data can be used, based on fragmenting the data and storing only relevant parts in memory such as Apache Axiom model based on a novel ‘pull-through’ parsing method [2]. Such methods could overcome the memory limitations of DOM-based processors and at the same time introduce clear design into streaming processors.

References