Warp-Edge Optimization in XPath

Haiyun He and Curtis Dyreson

School of EE and Computer Science
Washington State University, Pullman, WA 99164-2752
{hhe, cdyreson}@eecs.wsu.edu
http://eecs.wsu.edu/~cdyreson

Abstract. We describe the design and preliminary analysis of an optimization technique for XPath called *warp-edge optimization*. The XPath data model is a tree-like data model that has an edge from an element to each component in the content of that element. The edges are traversed in the evaluation of an XPath expression. A *warp edge* is an edge that is something other than a parent to child edge, i.e., an edge from an element to a sibling or to a grandchild. Warp edges can be dynamically generated and stored during query evaluation to improve the efficiency of future queries. We describe the implementation of warp-edge optimization as a layer on top of Xalan, the XPath evaluation engine from Apache. Experiments demonstrate that in the evaluation of some XPath expressions, the use of warp edges results in substantial savings of time.

1 Introduction

The explosive growth of the World-wide Web (web) has lead to an increase in the number of organizations that use the Extensible Markup Language (XML) to exchange data [1]. XML is a markup language for specifying the structure and semantics of text data and documents. XML avoids common pitfalls in language design, is extensible, platform-independent, and supports internationalization [2].

There are several query languages for XML data collections. Examples include Lorel [3], XQuery [4], XML-QL [5], and XSL Transformations (XSLT) [6]. An important component in many of these languages, especially those promulgated by the W3C, is XPath [7]. XPath is a language for addressing parts of an XML document. For instance the XPath expression '`(//paragraph)[5]`' locates the fifth paragraph element in a document. XPath expressions are a core component of all XSLT and XQuery programs.

In this paper, we propose an optimization technique for XPath called *warp-edge optimization*. The XPath data model is a tree-like data model that has an edge from an element to each component in the content of that element. The edges are traversed in the evaluation of an XPath expression. A *warp edge* is an edge that is something other than a parent to child edge, i.e., an edge from an element to a sibling or to a grandchild. Warp edges can be generated and stored during query evaluation to improve the efficiency of future queries. In the evaluation of some XPath expressions, the use of

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

warp edges results in substantial savings of time since the warp edges connect nodes separated by two or more non-warp edges.

This paper makes several contributions. First, we discuss how to support warp-edge optimization by dynamically caching query results. Second, we implement the technique as a *layer* on top of, but separate from, an XPath evaluation engine. The important advantage offered by a layered architecture is that the warp-edge optimization layer can be combined with any XPath evaluation engine. Hence, we do not have to modify an XPath evaluation engine to optimize XPath queries. Third, we report on some experiments that demonstrate the efficacy of the technique.

The remainder of this paper is organized as follows. In the next section we motivate the technique. We then briefly sketch the implementation of the technique and experimental results.

2 Motivation

In this section, an example is provided to demonstrate warp-edge optimization. Consider a sample XML document shown in Figure 1. The fragment shows part of a novel, and is short for expository purposes. The document root is the `<doc>` element. Within the root are a title and a chapter. The chapter also has a title and has several sections. Each section has a title, a paragraph and a note.

```
<doc>
  <title>A Tale of Two Cities</title>
  <chapter>
    <title>The Journey</title>
    <section>
      <title>The Beginning</title>
      <para>It was the best of times…</para>
      <note>A famous opening line.</note>
    </section>
    <section>
      <title>The Middle</title>
      <para>The second city was
        <emph>squalid</emph> in a tepid way.
      </para>
      <note>This not quite so famous.</note>
    </section>
  </chapter>
</doc>
```

Figure 1 A sample XML document

The XPath data model for the sample document is depicted in Figure 2. The data model is constructed when the document is parsed. Details of the model extraneous to the example have been omitted (e.g., text nodes); only element nodes are shown. The "id" of the node is shown within the node.

Assume that a user of the digital library retrieves all of the sections by submitting the query "`//section`". The warp edges created by the query are shown as dashed

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

lines in Figure 2. The warp edges connect the root with each section since the query starts at the document root and terminates at each section. The size of the sample document is small and the sections can be found quickly, but in general the document could contain thousands of sections.

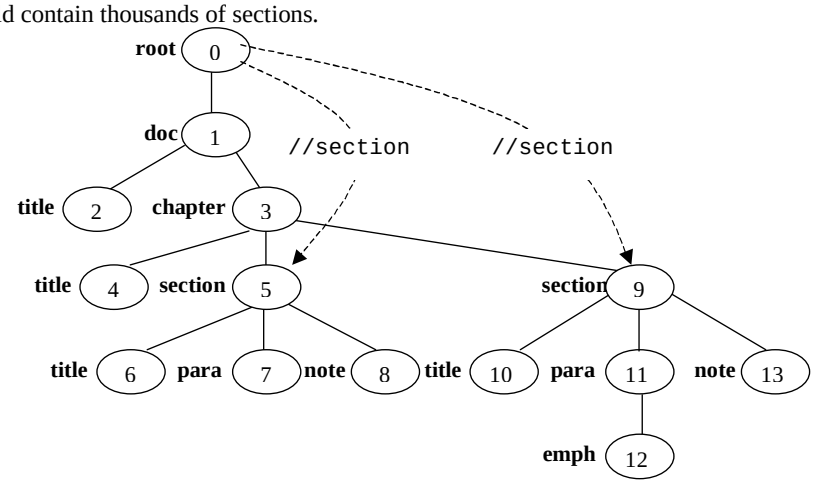

Figure 2 The data model for the sample (warp edges are dashed lines)

XSLT and XQuery programs usually contain many XPath expressions. Assume that queries are subsequently given to retrieve the section titles, "`//section/title`", and to retrieve the chapters that contain sections, "`//chapter[section]`". Both queries can make use of the warp edges. The first query can traverse the warp edges to locate sections quickly, and then drop down to the title nodes. By using the warp edges the four edges on the path to each section node can be skipped. Not a big savings, but this document is quite small. Section 4 demonstrates the effectiveness of the technique on large trees. The second query can warp to sections, and then move up to find chapter nodes. This will be unlikely to result in a faster evaluation since four edges need to be traversed using both evaluation strategies (with and without warp edges). Sometimes following warp edges does not save time.

Related Work

There are two common technologies for query optimization in semistructured databases and XML query languages. The first is to build performance-enhancing data structures, e.g., indexes, and generate a query evaluation plan utilizing the structures. Lore has several indexes, such as value and path indexes [8]. Lorel queries can be compiled into plans that make efficient use of the indexes [9]. Other path indexes include the t-index [10] and the Index Fabric [11]. For XPath, the Dynamic XML Engine (DXE) takes advantage of available indexes to accelerate queries [12]. Warp-edge optimization is similar because it builds a "path index" consisting of the warp

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

edges. However, the index is constructed on-demand and in an ad hoc manner, unlike a DataGuide [13]. The above systems (except DXE) are database systems where the cost of statically building indexes is small in comparison to the benefits, whereas warp-edge optimization is applicable to in-memory XML parsing.

The second common technology is to rewrite the query (or batch of queries) to prune the search space. Gardarin, Gruser and Tang propose a technique to optimize linear path expressions and produce a cheap query execution plan [14]. Compile-time path expansion [15] utilizes a DataGuide (a schema) to prune the search space, while branching path optimization [16] recognizes that queries that follow the same branch in a tree can share the cost of exploring that branch, as does warp-edge optimization. Optimization in StruQL [17] combines both indexing and query rewriting. Warp-edge optimization dynamically implements branching path optimization.

3 Warp-edge Optimization

A warp edge is an edge in the data model that traverses more than one level in the document tree. The canonical example is an edge to a grandchild node. Typically, warp edges are added as the result of previous queries. The warp edges can be traversed during the evaluation of a query

3.1 An Optimization Layer

There are two basic strategies for implementing warp-edge optimization. One approach is to modify the query evaluation engine to add warp edges to the data model. The second approach is to add a *layer* to perform warp-edge optimization above the legacy system. We adopt the layer approach because it is more flexible when the underlying system changes and can be implemented on proprietary evaluation engines. Any legacy system could be used such as Saxon [18], Sablotron [19], XT [20], Microsoft's XML Core Services (MSXML) 4.0, or Xalan [21]. Figure 3 depicts the layered approach with the Xalan, the XPath evaluation engine from Apache. The warp-edge layer sits on top of the Xalan package. The layer takes a query and splits it into several sub-queries. Some of the sub-queries can be answered from the results of previous queries that are stored in the layer in an area called the *query cache*. The sub-queries that cannot be answered are sent to Xalan for evaluation. Xalan itself is not modified in any way, rather the layer uses Xalan's API. The layer processes the results of the sub-queries to build the result set. The results of sub-queries that generate new warp edges are stored in the query cache.

Caching the query results *induces* a set of warp-edges in the underlying data model. Whenever a new query result is added to the cache, in effect, it represents a corresponding warp edge in the parsed document. The opportunity to reuse query results increases as more results are added to the query cache.

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

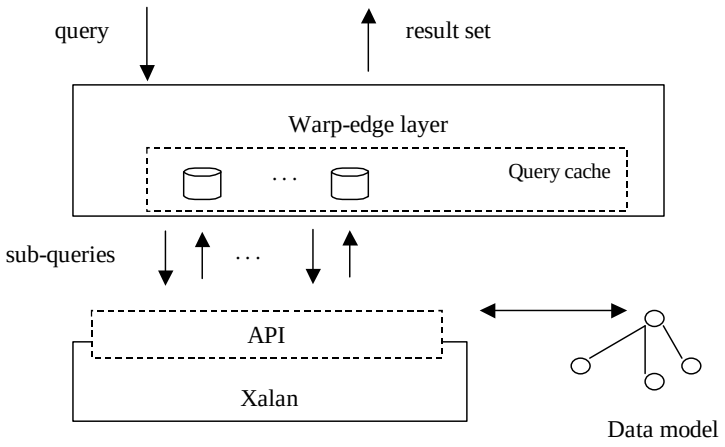

Figure 3 Layer approach with Xalan

3.2 Prefix Matching

A query is evaluated by trying to find the *longest matching prefix* in the query cache. To facilitate the matching, the query cache is organized as a collection of trees called query cache trees (QCTs). When a query is issued, the layer first looks for the query in the cache. If the result is already there, the result is returned immediately because a new XPath query might be the same as an old one. On the other hand, a new XPath query can be different from all old queries, but we can still take advantage of the cached query results. The trick here is that we can use the cached result to "construct" the result for a new XPath query. For example, the new query may be an extension of an old query or may contain a part that has been evaluated before. The cached result is not returned immediately. Instead, it can be used as a temporary result to facilitate the evaluation of new query.

There are three outcomes to the prefix match.

1) *Full Cache Hit*: If the entire query is matched then the query is totally the same as a previous query and the result is already available.
2) *Partial Cache Hit*: A prefix of the query (i.e., the first few steps) match, then the cached prefix becomes the *context* for further evaluation of the remaining steps in the query. Prefix matching is performed for the rest of the steps in the query for each node in the context.
3) *Cache Miss*: This happens when a new query starts with a different step than all previous queries. A single step in the query is evaluated to establish a context for subsequent steps. Then the prefix match is tried against the remaining steps in the query.

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

At worst, the query is evaluated one step at a time by evaluating each step on the underlying query engine (every step results in a cache miss). Ideally, sequences of one or more steps can be found in the query cache (a cache hit). Then the cached result can be used without consulting the underlying query evaluation engine.

4 Empirical Analysis

In this section, we describe a preliminary set of experiments. Our goal is to test warp-edge optimization to determine whether it works under "ideal" conditions. The experiments involve tests on randomly generated data. We describe the parameters of each experiment in detail. Finally, we analyze the results.

4.1 Experimental Environment

We conducted the experiments on a Pentium PC (Dell Precision 340). It has an Intel® Pentium® 4 CPU 1700MHz, 512MB RAM and 37.2GB disk space. The PC runs Windows XP Professional Version 2002. We installed Java™ 2 v1.3.1_02 and Xalan-Java v2.3.1 for testing. The XML Parser used is Xerces-Java v2, which is available with the Xalan-Java package. We isolated the machine for testing. Only the test program and normal background processes are running during the testing period.

4.2 Random Experiment

We generated random XML documents for testing with the following configurable parameters.

- The children of root factor – This factor represents the number of children of the document root. It controls the top-level bushiness of the XPath data model tree.
- The depth factor – This factor represents the level of nesting of elements in the XML document. It controls the depth of the data model tree.
- The bushy factor - This factor describes the number of children in a non-leaf node in the data model tree. The bushiness can be fixed or chosen randomly from a range.

The tree is made random in two ways. First, the depth and bushiness of the tree can be made random to test with short, busy trees or deep, skinny trees, or some combination thereof. Because of limited memory, the trees are capped in size at approximately 12,000,000 nodes. Second, each level in the tree consists (almost) entirely of the same kind of elements, e.g., level one consists of `<A>` elements, level two of `<B>` elements, etc. However, we randomly convert up to 10% of the elements at each level into "magic" elements; a magic element is appended with a number e.g., `<B1>`. In a

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

query, the magic elements can be used for node tests to limit the result-set size, e.g., the query '`//B/C`' will return far more nodes than '`//B1/C1`'.

4.2.1 High match probability experiment

In this experiment, we tested the performance using XPath queries that have high match probabilities, i.e., there is a greater chance to retrieve a large result set. We tested the following query batches on the randomly generated XML documents. The query cache is updated after each batch.

```
Batch 1:    /descendant-or-self::C
            /descendant-or-self::E

Batch 2:    /descendant-or-self::B/child::C
            /descendant-or-self::D/child::E

Batch 3:    /descendant-or-self::C/child::D
            /descendant-or-self::C/descendant-or-self::E
            /descendant-or-self::C/descendant-or-self::F
```

The above query batches only include "pure elements" and therefore have large result-sets. This simulates the situation where a user requests popular information from an XML document. Furthermore, the batches are designed to favor warp-edge optimization since the last two query batches utilize the warp edges. We tested a range of root children, depth factors and bushy factors independently, and averaged the results of the tests on runs of five random trees.

In the first experiment, we varied the number of root children and fixed the depth factor to be 6 and bushy factor to be 3. The turnaround time result is shown in the left-hand graph in Figure 4 (including the time to update the QCT). The right-hand graph shows the space overhead of the QCT.

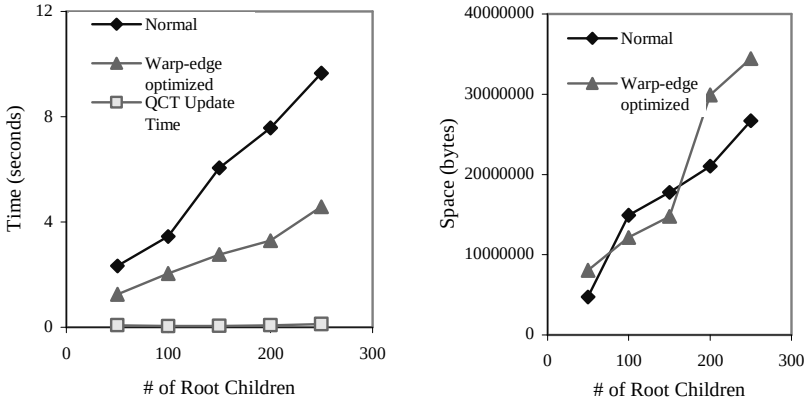

Figure 4 Varying the number of root children

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

Next, we varied the depth factor but fixed the number of root children to be 50 and the bushy factor to be 3. Then the XML document is moderately bushy with a moderate number of sub-trees, but varies from shallow to deep. We obtain the graphs in Figure 5

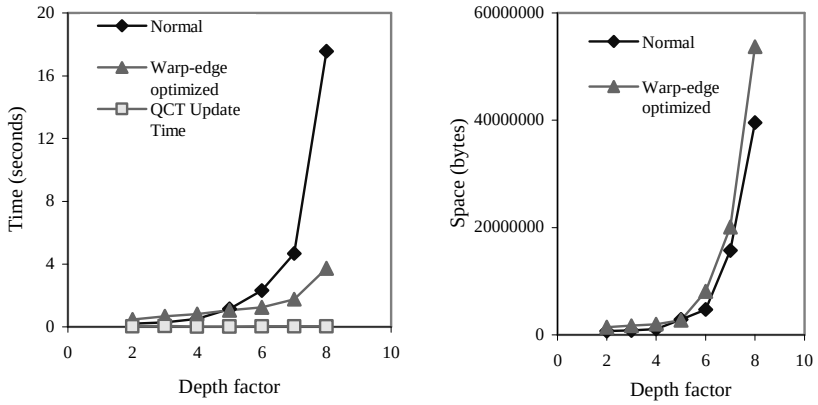

Figure 5 Varying the depth factor

Third, we varied the bushy factor and fixed the other parameters, i.e. the number of root children is 50 and the depth factor is 5. Then the XML document will have a moderate number of sub-trees and be of moderate depth, but will vary in bushiness from skinny to fat trees. By this means, we can see how our approach performs with a change in bushiness. The results are shown in Figure 6.

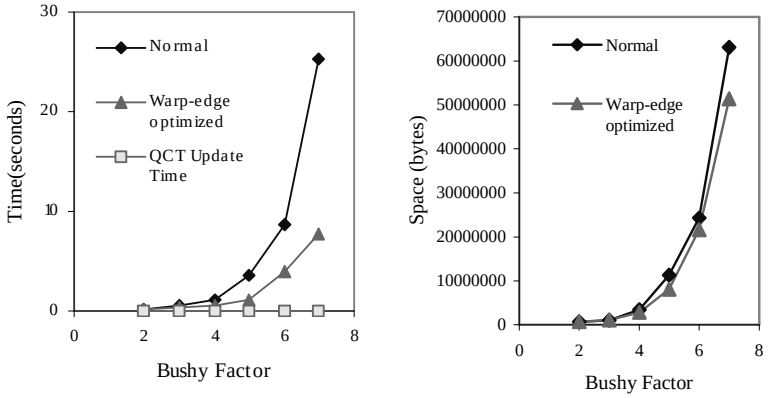

Figure 6 Varying the bushy factor.

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

The turnaround time for query evaluation shows that the optimization is working best for deeper and bushier trees. In the first experiment, the optimization approximately halves the time needed for query evaluation at a modest increase in the amount of space. In the second experiment, although query performance degrades exponential (as the size of the tree increases exponentially), the non-optimized query time increases much faster than that of the optimized query. For very deep trees, when the depth reaches 8, the optimization provides a five-fold increase in query performance. Again, only a small amount of additional space is needed for the optimization. The third experiment, testing trees of varying bushiness, confirms that the optimization can improve query performance when more warp edges are utilized in the larger and bushier trees.

The graphs also depict the time needed to update the query cache trees (QCT). The update time is not counted in the turnaround time. The cost however, is usually quite trivial. The reason is that the QCT update just generates a mapping between the query and the corresponding result set, which is not a time-consuming process.

Overall, the experiments show that while warp-edge optimization needs a small amount of additional space, it can improve query performance for large, deep, and bushy trees.

5 Conclusions and Future Work

In this paper we described the design and analysis of an optimization technique for XPath called warp-edge optimization. Warp edges can be dynamically generated and stored during query evaluation to improve the efficiency of future queries. We implemented warp-edge optimization as a layer on top of Xalan, the XPath evaluation engine from Apache. Experiments demonstrate that in the evaluation of some XPath expressions, the use of warp edges results in substantial savings of time at a modest increase in space. The benefit of the layered implementation is that warp-edge optimization can be wrapped around any back-end XPath evaluation engine. Our experiments show that the cost of the layer is small.

In future, we plan to develop query rewrite rules to support more effective use of the cache in a manner similar to rewriting database queries using materialized views. Also, since the cache independently maintains some information, we believe that query caching can be used to provide partial answers when the original document is no longer available or expensive to query directly.

References

1. World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3c.org/XML. Current as of October 2000.
2. World Wide Web Consortium. XML in 10 points. http://www.w3c.org/XML/1999/XML-in-10-points. Current as of November 2001.

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.

3. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data**.** SIGMOD Record, 26(3):54-66, September 1997.
4. World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3c.org/TR/xquery/. Current as of April 2002.
5. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. WWW10, Toronto, CA.
6. World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0. http://www.w3c.org/TR/1999/REC-xslt-19991116. Current as of November 1999.
7. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3c.org/TR/xpath. Current as of April 2002.
8. J. McHugh and J. Widom. Query Optimization for XML. In Proceedings of VLDB, Edinburgh, Scotland, September 1999.
9. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical Report, Stanford University, Database Group, January 1998.
10. T. Milo and D. Suciu. Index structures for path expressions. In ICDT'99, Jerusalem, Israel, January 10-12, 1999, pages 277-295, 1999.
11. B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In Proceedings of VLDB, September 2001, pp. 341-350.
12. Exceloncopr. Optimizing XPath Expressions. http://support.exceloncorp.com. Current as of May 2001.
13. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In Proceedings of VLDB, August 1997, pp. 436-445.
14. G. Gardarin, J.Gruser, and Z. Tang. Cost-based Selection of Path Expression Processing Algorithms in Object-oriented Databases. In Proceedings of VLDB, Bombay, India, pp. 390-401.
15. J.McHugh and J. Widom. Compile-Time Path Expansion in Lore. In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.
16. J. McHugh and J.Widom. Optimizing Branching Path Expressions. Technical report, Stanford University, Database Group, June 1999.
17. M. Fernandez, D. Florescu, J.Kang, A. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-site Management System. In Proceedings of SIGMOD, Seattle, Washington, June 1998, pp. 414-425.
18. Michael Kay. SAXON The XSLT Processor. http://saxon.sourceforge.net. Current as of February 2002.
19. Ginger Alliance. Sablotron XSLT, DOM and XPath processor. http://www.gingerall.com/charlie/ga/xml/p_sab.xml. Current as of March 2002.
20. James Clark. XT. http://www.jclark.com/xml/xt.html. Current as of November 1999.
21. Apache XML Project. Xalan-Java version 2.3.1. http://xml.apache.org/xalan-j/index.html. Current as of March 2002.

Lecture Notes in Computer Science 2426, Advances in Object-Oriented Systems, Proceedings of EWIS 2002, Montpellier, France, September, 2002, pp. 187-196.