Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts.

Jonathan Lee Leonard
East Tennessee State University

Follow this and additional works at: https://dc.etsu.edu/etd
Part of the Theory and Algorithms Commons

Recommended Citation

This Thesis - Open Access is brought to you for free and open access by the Student Works at Digital Commons @ East Tennessee State University. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ East Tennessee State University. For more information, please contact digilib@etsu.edu.
Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts

A thesis
presented to
the faculty of the Department of Computer and Information Sciences
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Master of Science in Applied Computer Science

by
Jonathan Leonard
May 2006

Dr. Phillip Pfeiffer, Chair
Dr. Don Sanderson
Dr. Chris Wallace

Keywords: XML, RDBMS, ORDPATH Encoding, Interval Encoding
ABSTRACT

Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts

by

Jonathan Leonard

The rise of XML as a de facto standard for document and data exchange has created a need to store and query XML documents in relational databases, today’s de facto standard for data storage. Two common strategies for storing XML documents in relational databases, a process known as document shredding, are Interval Encoding and ORDPATH Encoding. Interval Encoding, which uses a fixed mapping for shredding XML documents, tends to favor selection queries, at a potential cost of $O(N)$ for supporting insertion queries. ORDPATH Encoding, which uses a looser mapping for shredding XML, supports fixed-cost insertions, at a potential cost of longer-running selection queries. Experiments conducted for this research suggest that the breakeven point between the two algorithms occurs when users offer an average 1 insertion to every 5.6 queries, relative to documents of between 1.5 MB and 4 MB in size. However, heterogeneous tests of varying mixes of selects and inserts indicate that Interval always outperforms ORDPATH for mixes ranging from 76% selects to 88% selects. Queries for this experiment and sample documents were drawn from the XMark benchmark suite.
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>2</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>5</td>
</tr>
<tr>
<td>LIST OF SOURCE CODE LISTINGS</td>
<td>6</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>7</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>7</td>
</tr>
<tr>
<td>2.1 XML and XQuery</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 XML</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 XQuery and Data Adapters</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Document Encoding for XML Documents</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Order Encoding Methods</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Interval and Dynamic Interval Encoding</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3 ORDPATH Encoding</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Summary</td>
<td>22</td>
</tr>
<tr>
<td>3. EXPERIMENTAL METHODS</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Motivation and High Level Strategy</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Low-Level Strategy and Implementation Details</td>
<td>28</td>
</tr>
<tr>
<td>4. DATA</td>
<td>33</td>
</tr>
<tr>
<td>5. ANALYSIS</td>
<td>43</td>
</tr>
<tr>
<td>5.1 Qualitative Analysis</td>
<td>43</td>
</tr>
<tr>
<td>5.2 Quantitative Analysis</td>
<td>43</td>
</tr>
<tr>
<td>5.2.1 Shred Comparison</td>
<td>43</td>
</tr>
<tr>
<td>5.2.2 Query Performance Comparison</td>
<td>44</td>
</tr>
</tbody>
</table>

3
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1. Order Encoding Methods</td>
<td>16</td>
</tr>
<tr>
<td>2-2. Interval Encoding of Listing 2-2</td>
<td>18</td>
</tr>
<tr>
<td>2-3. ORDPATH Encoding of Listing 2-3</td>
<td>20</td>
</tr>
<tr>
<td>2-4. ORDPATH Prefix Mapping Table</td>
<td>21</td>
</tr>
<tr>
<td>4-1. Running Times for Homogeneous Benchmarking (in Wall seconds)</td>
<td>33</td>
</tr>
<tr>
<td>4-2. Encoded File Sizes (in bytes)</td>
<td>34</td>
</tr>
<tr>
<td>4-3. Running Times for Heterogeneous Mixes of Benchmarks</td>
<td>34</td>
</tr>
<tr>
<td>4-4. Interval Query Performance</td>
<td>34</td>
</tr>
<tr>
<td>4-5. ORDPATH Query Performance</td>
<td>35</td>
</tr>
<tr>
<td>4-6. Query Comparison – XMark 0.01 Document</td>
<td>35</td>
</tr>
<tr>
<td>4-7. Query Comparison – XMark 0.02 Document</td>
<td>36</td>
</tr>
<tr>
<td>4-8. Query Comparison – XMark 0.03 Document</td>
<td>36</td>
</tr>
<tr>
<td>4-9. Query Comparison – Average</td>
<td>37</td>
</tr>
<tr>
<td>4-10. Interval Insert Performance</td>
<td>37</td>
</tr>
<tr>
<td>4-11. ORDPATH Insert Performance</td>
<td>38</td>
</tr>
<tr>
<td>4-12. Insert Comparison – XMark 0.01 Document</td>
<td>38</td>
</tr>
<tr>
<td>4-13. Insert Comparison – XMark 0.02 Document</td>
<td>39</td>
</tr>
<tr>
<td>4-14. Insert Comparison – XMark 0.03 Document</td>
<td>39</td>
</tr>
<tr>
<td>4-15. Insert Comparison – Average</td>
<td>40</td>
</tr>
<tr>
<td>4-16. Heterogeneous DB Operation Comparison</td>
<td>40</td>
</tr>
<tr>
<td>4-17. Encoding Size Comparison – XMark 0.01 Document</td>
<td>41</td>
</tr>
<tr>
<td>4-18. Encoding Size Comparison – XMark 0.02 Document</td>
<td>41</td>
</tr>
<tr>
<td>4-19. Encoding Size Comparison – XMark 0.03 Document</td>
<td>42</td>
</tr>
</tbody>
</table>
# LIST OF SOURCE CODE LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1. XML Example</td>
<td>12</td>
</tr>
<tr>
<td>2-2. XML Fragment</td>
<td>18</td>
</tr>
<tr>
<td>2-3. Another XML Fragment</td>
<td>20</td>
</tr>
<tr>
<td>3-1. Example auction.xml</td>
<td>23</td>
</tr>
<tr>
<td>3-2. AbstractTraverser Interface Specification</td>
<td>27</td>
</tr>
<tr>
<td>3-3. Interval-Specific Implementation of getChildrenByName</td>
<td>28</td>
</tr>
<tr>
<td>3-4. Interval-Specific Implementation of Query 3</td>
<td>29</td>
</tr>
<tr>
<td>3-5. ORDPATH-Specific Implementation of getChildrenByName</td>
<td>30</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

This work investigates the performance of methods for storing and querying XML documents in relational databases. XML has become the de facto standard for document and data exchange on the Internet. XML’s popularity is due in large measure to its expressiveness: XML’s expressive data representation and query language support the use of arbitrarily extensible hierarchical structures for capturing taxonomy-based content, a common format for representing information [1].

The rise of XML has generated interest in native XML storage and retrieval systems. These systems, which were first developed in the late 1990’s, are designed for storing and retrieving XML documents, and are not simply database systems with XML layers on top. One of the most popular commercial products is Software AG’s Tamino, which was first released in 1999 [1]. Well-known open source native XML databases include eXist\(^1\) and Apache’s Xindice\(^2\). Web administrators who have considerable amounts of XML to store and retrieve are the most common users of these systems. Xindice is probably more popular for Web applications, due to its integration with Apache’s Tomcat Server.

Even with this interest in native XML storage, the maturity and widespread deployment of relational database technologies suggests a need for storing XML documents in relational databases. Relational database management systems (RDBMSes) have been in use over 20 years and are the de facto standard database implementation. Mature RDBMSes like Oracle, SQLServer, PostgreSQL, and MySQL are more common than native XML databases, and they feature enabling technologies like sophisticated query optimizers that native XML systems still lack. RDBMSes also scale well, perform well, and have proven resilient over time: when object-oriented database management systems (OODBMS) were introduced in the early 1990’s, RDBMSes were adapted to emulate OODBMSes, and performed as efficiently as native OODBMSes, due to their 20+ years of query optimization.

The literature on RDBMS support for XML describes various strategies for document shredding: i.e., the encoding (i.e., storing) of an XML document in an RDBMS. The primary focus of this thesis is a performance comparison between two related strategies for document shredding: Interval Encoding, a.k.a. Nested Sets [3], and, ORDPATH Encoding, which is used in the latest version of Microsoft® SQL Server™ [4]. The two methods share important similarities that make them interesting candidates for performance evaluation. Neither method uses a separate schema to manage stored documents. Both methods, rather, map most XML nodes to a set of rows in RDBMS tables, with one set of columns used to capture information on document structure, and a second to capture content. Minor departures from this use of row-oriented embeddings mainly involve complex elements. These mappings, moreover, are order-preserving: they ensure the relative ordering of a document’s nodes when that document is retrieved from an RDBMS—in practice, by assigning an ordering identifier to each node.

Where the two kinds of methods differ is in their approach to encoding structure. Interval Encoding essentially assigns a global unique ID to each node in an XML document as it scans from top to bottom. In addition to this ID, another value is assigned to each node that represents the width of the forest rooted at that node. ORDPATH, on the other hand, uses a Dewey-decimal numbering system that assigns to each node the materialized path to that node. The net effect of these differences is that Interval Encoding and ORDPATH should perform differently, relative to operations that query and update a database. Specifically, Interval should be faster at performing selects and ORDPATH should be faster at performing inserts. ORDPATH is a O(C) algorithm for inserts, though lengthening paths can eventually affect performance, and Interval is a O(N) algorithm for inserts.

Data on the relative performance of Interval and ORDPATH encoding has not been published with respect to a standard benchmark. The work described here sought to address this gap in the literature, and to determine when it would be better to use the insert-friendly ORDPATH encoding instead of the select-friendly Interval Encoding algorithm. To achieve this goal, the 20 queries that make up the
XQuery-based XMark benchmark were used to compare the algorithms’ performance, relative to a series of sample datasets. These comparisons sought to determine a proportion of insertion to selection operations that put ORDPATH on an equal footing with Interval Encoding. Users whose query mixes featured a greater and lesser proportion of insertions would be advised to use ORDPATH and Interval encodings, respectively.

The work included two implementations of the XMark benchmark suite, relative to two libraries of Python routines that supported query implementation using ORDPATH and Interval Encodings. These supporting libraries used a common interface, which allowed sample queries to be coded once. The library contains a set of path-oriented data access functions including getRootNode and getNodeAtXPath as well as a set of traversal-oriented data access functions including getFirstChild and getNextSibling. Functions were added to this common interface as they were found to be needed.

Python was selected for its rapid development capabilities and platform-independence. In practice, more efficient running times could be achieved by using a compiled language; however, since both methods were handicapped by the same amount, this should not have affected the comparisons.

All benchmarks were run on a Celeron 2.2 GHz processor with 512 MB of RAM and utilized the MySQL database. Test XML files of sizes 1.5 MB, 3.0 MB and 4.5 MB were used.

The data from these experiments shows that queries ran on average 6.38 times slower and inserts ran approximately 31.16 times faster for ORDPATH Encoding than for Interval Encoding. Accordingly, for XML files of sizes in the range tested, it would be beneficial to implement ORDPATH Encoding only if less than 5.6 select queries per insert are expected.

The remainder of this work includes detailed chapters on background information, including detailed descriptions of Interval and ORDPATH Encoding; a discussion of the experiments used to compare Interval- and ORDPATH-encoded documents; the results of these comparisons; an analysis of these results; and a final section that presents conclusions, and suggests further avenues for work.

---

CHAPTER 2

BACKGROUND

This chapter includes background information and resources covering XML and XQuery as well as several methods for XML document encoding, which is also known as shredding.

2.1 XML and XQuery

2.1.1 XML

The Extensible Markup Language (XML) is a universal format for documents and data shared on the Internet. XML was initially developed by the World Wide Web Consortium (W3C) in 1996 by a group chaired by Jon Bosak of Sun Microsystems. XML, a subset of the Generic Standard Generalized Markup Language (SGML, ISO 8879:1986), was designed for interoperability with HTML. An SGML Working Group, also organized by the W3C, assisted with the development of XML. The initial goal of these groups was to allow for SGML to be “served, transmitted and processed” on the Web as easily as HTML. W3C published the resulting work, the W3C XML Recommendation, in 1998. Two versions of the recommendation have been released since 1998 [8].

SGML was initially conceived in the 1960’s and 1970’s and became an ISO standard (ISO 8879) in 1986. SGML and XML are markup languages: they support the coding of human-readable data and the metadata that describes it. Valid XML documents are valid SGML documents with additional restrictions that make these documents easier to parse. Some applications of SGML and XML include Hypermedia/Time-based Structuring Language (HyTime), Standard Music Description Language (SMDL), and the News Industry Text Format (NITF) [8]. These languages are best described as applications or instances of SGML or XML. XML and SGML allow users to define grammars for particular applications, including the tags that define a language’s vocabulary. Specifying a vocabulary of tags brings an instance of an XML language into existence.

XHTML, a proposed replacement for HTML, is one important example of an XML-based vocabulary. HTML, which was developed before XML, uses harder-to-parse features of SGML like
support for unclosed tags that were not incorporated into XML. XHTML eliminates these features while extending HTML in two key ways: XHTML allows its users to define new tags and to decouple a web page’s content from its appearance in a way that allows formatting to be managed by a device-specific rendering agent. This is quite advantageous as more diverse devices are used to browse the Web (such as telephones, PDAs, and televisions). These devices may choose how to display the structured data according to rules applicable to their domain through the use of stylesheets (XSLT or CSS transformations). Another design goal of XHTML was internationalization. In short, XHTML tries to move to a more general expression of web page data.

XML documents can be data- or document-centric. A document-centric XML document consists of free-form text that is “marked-up” with elements that specify italic, bold, and other usual presentation-oriented metadata. A data-centric document, on the other hand, consists mostly of data with elements specifying the dimensions of the data (such as length and height).

An XML document consists of storage units known as entities that may contain either parsed or unparsed content. Parsed content consists of character strings that are classified as either markup, unparsed data, or parsed data. Markup specifies a document’s logical structure and storage layout. Unparsed data is left to the application to process. The final type of parsed content, parsed data, is parsed by XML Processors [8]: parsers that read XML documents and provide access to their content and structure. Simple API for XML (SAX) is one of the most common XML Processors. SAX was initially developed for Java but has been ported to several other languages, including Perl and Python.

XML provides a better metaphor for modeling information than preceding data-modeling mechanisms. It does so by supporting the development of heterogeneous, extensible, and flexible data models. Heterogeneity is the ability to express records that may contain different fields. The XML notion of a record allows for varying numbers of fields (or elements in XML terminology). Extensibility is the ability to add new types of data at will without determining them in advance. XML allows for fields with flexible size and configuration from instance to instance. Each data element can be as long or as short as it needs to be [1].
XML is also informationally complete and self-describing: the structure of a program’s data can be specified in XML and operated on using preexisting logic. Microsoft and other companies offer frameworks for building applications based on using XML to express information. As Chaudhry notes, “In environments such as these, XML becomes a universal information-structuring tool where system components no longer need to be programmed separately as discreet silos of functionality [1].” Thus, when using an XML-centric system, one can often accommodate changes to a dataset by modifying the underlying XML and letting the application adjust itself accordingly.

There are four basic components used in expressing information in XML: tags, attributes, data elements, and hierarchy. Each component represents a different dimension of information. A code sample will suffice to convey an understanding of these basic components:

<table>
<thead>
<tr>
<th>Listing 2-1: XML Example [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;colorimeter_reading&gt;</td>
</tr>
<tr>
<td>&lt;RGB resolution=8&gt;</td>
</tr>
<tr>
<td>&lt;red&gt; 0 &lt;/red&gt;</td>
</tr>
<tr>
<td>&lt;green&gt; 1 &lt;/green&gt;</td>
</tr>
<tr>
<td>&lt;blue&gt; 255 &lt;/blue&gt;</td>
</tr>
<tr>
<td>&lt;/RGB&gt;</td>
</tr>
<tr>
<td>&lt;/colorimeter_reading&gt;</td>
</tr>
</tbody>
</table>

In this example, data elements are represented in bold. A data element is the same as the traditional idea of data. XML adds meaning to the data by adding tags (the values specified inside <>’s). Tags describe what the data elements are—i.e., tags are metadata. Attributes, specified in italics in the example, support the interpretation of data elements. Lastly, the hierarchy of elements is specified by containment [1].

When an element is contained by another element, it is a child of that element. An element that contains child elements is a complex element whereas one that does not is a simple element. All XML documents have at least one node or element that is the root of all other nodes.

An XML schema language describes an XML document’s structure and content. An XML schema describes the contract between the producer and consumer of an XML document. It essentially defines what constitutes a valid ‘XML message’ between the two parties. Formats for XML schemas
include DTD, the one inherited from SGML; XDR; and the W3C XML Schema Definition Language (XSD). Currently, XSD is the most popular [9].

XSD was the first XML schema language to expand the role of a schema outside its traditional role of describing the contract between the interchanging parties. It does this through the concept of a Post Schema Validation Infoset (PSVI). An XSD processor accepts as input an XML Infoset and produces as the result a PSVI upon validation. During processing the XSD processor assigns a type to each element and attribute parsed. Then, applications can retrieve type information from the PSVI. According to Obasanjo, “Such strongly-typed XML is very versatile because it can now be mapped to objects using technologies like the .NET Framework’s XMLSerializer, mapped to relational tables using technologies like SQLXML and the .NET Framework’s Dataset, or it can be processed using XML Query languages that take advantage of strong-typing, such XPath 2.0 and XQuery [9].”

2.1.2 XQuery & Data Adapters

As XML gains in popularity as the standard way to expose data online, the need for a query language for XML data is increasing. Several XML query languages are competing for general acceptance. Currently, XQL has the greatest market penetration. However, most experts expect XML Query, or XQuery, to become the de facto standard due to its rich feature set and endorsement by the W3C (World Wide Web Consortium). One indication of XQuery’s importance is the fact that IBM and Oracle have “collectively launched an effort to connect XQuery to Java [10].”

The W3C began work on XQuery in 1998, releasing an initial standard in February 2001. Since that time, XQuery has gone through several revisions. Of the W3C specifications, XQuery is generally regarded as the most carefully designed and therefore has also been the slowest to evolve [10]. The slow evolution of XQuery has been attributed to the lack of prior experience in retrieving XML data. The XQuery specification became a W3C recommendation on 3 November 2005 [11].

The beauty of XML is that it can organize data in a hierarchical, object-oriented, and multi-dimensional way. An XML document represents data in the form of a tree with an arbitrary depth and
width. A traditional relational database table can be thought of as a tree of depth two with unbounded fanout at the first level, and fixed fanout at the second level with the first level representing tuples (or rows) and the second level representing fields (or columns). An XML tree is clearly a more expressive way of representing data as no constraints are placed on either depth or width.

XML’s support for expressive data modeling, however, will probably not be enough to motivate companies to replace existing relational databases with XML databases. XML does not yet compare favorably with relational technology in two important regards. First, neither XML nor object-oriented data stores scale as well, in terms of complexity, as relational systems [10]. Second, there is just too much already invested in RDBMS technology; until there is a “quantum leap in technology [10]”, there is just no incentive to replace RDBMSes. This point is made by Ivanov in “XQuery Implementation”, who notes that even if XML scaled and performed as well as relational databases, it would not be enough to “move the mountain of investment away from the RDBMS legacy [10].” Finally, there are certain classes of data that map well to the relational model and transitioning these to XML would increase overhead with no increase in functionality.

Given the continuing preeminence of RDBMSes, it becomes advantageous to make existing relational databases connect with the emerging XML data stores through use of a data adapter. A data adapter serves at least one and possibly two purposes. First, the necessary purpose is to allow an RDBMS to respond to queries formed against an XML representation of the data. This assumes that the person forming the queries knows the XML view of the data. If this is not the case, then the data adapter must additionally provide an XML view of the layout of the data against which proper queries can be formed.

A simple XML view of a relational database can be achieved by thinking of tables as trees as mentioned above. However, this view does not support transparent access to objects that have been broken into several tables due to normalization. Here, the data adapter must reassemble objects that have been normalized. This will allow queries to be written against the abstract view of the database (i.e., the view the database designer had in mind before it was normalized). In other words, the data adapter will infer joins from the query expression and return the results. A partial implementation of a data adapter
has been provided by this work. A query-supporting API will be provided for two popular methods of encoding XML documents in relational databases. This research stops short of a true data adapter by requiring manual coding of queries in terms of the API instead of allowing queries to be specified in terms of XQuery and automatically processed by a query interpretation layer.

A full appreciation for the power of XQuery depends, in part, on an understanding of XPath. XPath, another W3C specification, is a grammar for referencing parts of an XML document. XML-retrieval expressions supported by the XPath grammar resemble the use of path expressions for referencing the parts of a filesystem. An XPath expression consists of zero to many parent nodes and one leaf node, all separated by a specific delimiter. Additionally, XPath defines a set of standard functions for working with strings, numbers and Boolean expressions. XPath also defines ways for selecting unknown nodes, selecting multiple nodes that match certain criteria, selecting multiple paths and for selecting attributes of nodes among other things.

In its simplest form, an XQuery query can merely be an XPath expression. An XQuery query can also include static XML elements that are returned as is. While this is useful, the real power of XQuery lies in its FLWR (pronounced ‘flower’) expressions. FLWR stands for For-Let-Where-Return [11]. The FLWR expression allows for SQL-like SELECT functionality and can be fully nested. FLWR expressions also allow for the use of other expressions such as constructors and conditional and logical expressions. A set of XML elements is returned by FLWR expressions.

The main ingredients of a SELECT statement are SELECT, FROM and WHERE clauses. The SELECT portion specifies which columns to return. The set of tables to examine is specified by the FROM clause, and the WHERE clause specifies the search criteria. FLWR expressions are strikingly similar [11]. The For clause of FLWR specifies the XML node list to iterate over [11]. The Let clause allows for defining variables to use in the Where clause. The Where clause, as in SQL, specifies the search condition. Lastly, the Return clause specifies what exactly to return from the statement (allowing for restructuring of the data if needed). XPath expressions may be used in any of the four clauses.
XQuery is a functional programming language that contains standard functional operators, including arithmetic operators, comparison operators, and Boolean operators. It also contains four operators that apply specifically to sequences (item-at, except, union and intersect) and four operators that apply to nodes (node-after (>>, node-before (<<), node-equality (==), and node-inequality (!=)).

2.2 Document Encoding for XML Documents

2.2.1 Order Encoding Methods

The Order Encoding Methods are a set of 3 methods for representing XML node order in the relational database model. The first method, known as Global Order, assigns an increasing global index to each node as the document is scanned from top to bottom. The second, known as Local Order, assigns indices based on a node’s position relative to its siblings. The third, Dewey Order, is a hybrid encoding that assigns “a vector that represents the path from the document’s root to the node [2].” See Figure 1 for example encodings.

Figure 2-1: Order Encoding Methods [2]

<table>
<thead>
<tr>
<th>Global Order</th>
<th>Local Order</th>
<th>Dewey Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Global Order Encoding" /></td>
<td><img src="image2" alt="Local Order Encoding" /></td>
<td><img src="image3" alt="Dewey Order Encoding" /></td>
</tr>
</tbody>
</table>

**Global Order Encoding.** Global Order Encoding resembles the storing of lists in packed arrays. This encoding provides the best support of the three methods for order-related XML queries, including those that use order-axes, such as following and following-sibling. However, it also provides poor support for node insertion. At some point in a database’s lifetime, such as when a new node is inserted early in the document, many nodes will have to be renumbered. Inserts, on average, will force the renumbering of half the nodes in the database.
Sparse numbering (i.e., leaving gaps between nodes on inserts and deletes) reduces the need to renumber on inserts, but is not a magic bullet: eventually the gaps are used up and the problem reemerges. According to Tatarinov, “poor insertion performance is … a potential weakness of Global Order [2].”

Local Order Encoding. Local Order Encoding provides the best support for node insertions. With Local Order, a newly inserted node only requires renumbering the following siblings. Local Order Encoding, however, also provides the worst support for order-related queries. Global document order can be reconstructed from Local Order by combining a node’s position with those of its ancestors. However, combing a node’s position with those of its ancestors is a very expensive operation in SQL as there is no way to determine a given node’s ancestors without walking down to that node from the very top of the document. For this reason, Local Order Encoding has very limited practical use.

Dewey Order Encoding. Dewey Order Encoding strikes a balance between Global and Local Orders. According to Tatarinov, “query processing in Dewey Order is similar to that in Global Order.” When a new node is inserted at node X, renumbering is limited to X’s successor siblings and their descendants. Dewey Order provides many of the advantages of Global and Local Order, but it also requires extra space to store entire paths from the root to each node [2].

2.2.2 Interval and Dynamic Interval Encoding

Interval Encoding assigns each node an L value and an R value, according to the following rules:

- L < R for each node
- If node s₁ is an ancestor of node s₂ then L_s₁ < L_s₂ and R_s₂ < R_s₁
- If node s₁ is a left sibling of node s₂ then R_s₁ < R_s₂

An example of an XML document and its Interval encoding are shown Listing 2-2 and Figure 2-2. Interval Encoding is essentially Global Order Encoding with an additional R column, which characterizes the width of an XML forest: i.e., a subtree consisting of a node and its descendants. The R-column is particularly useful for narrowing the search domain when looking for specific nodes with given paths.
Also, the R-column makes iterating through siblings much easier and more efficient. The primary
disadvantage is the additional storage for the R-column.

Listing 2-2: XML Fragment [3]

```
<site>
  <people>
    <person id="person0">
      <name>Jaak Tempesti</name>
      <emailaddress>mailto:Tempesti@labs.com</emailaddress>
      <phone>+0 (873) 14873867</phone>
      <homepage>http://www.labs.com/~Tempesti</homepage>
    </person>
    ...  
  </people>
</site>
```

Figure 2-2: Interval Encoding of Listing 2-2 [3]

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;site&gt;</td>
<td>0</td>
<td>85</td>
</tr>
<tr>
<td>&lt;people&gt;</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>&lt;person&gt;</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>@id</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Person0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>&lt;name&gt;</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Jaak Tempesti</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>&lt;emailaddress&gt;</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td><a href="mailto:Tempesti@labs.com">mailto:Tempesti@labs.com</a></td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>&lt;phone&gt;</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>+0 (873) 14873867</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>&lt;homepage&gt;</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td><a href="http://www.labs.com/~Tempesti">http://www.labs.com/~Tempesti</a></td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

Several efficient algorithms for searching a document follow naturally from the addition of the R-
column. For instance, to determine the ancestors of a given node, select all nodes with lesser L values
and greater R values. Likewise, to determine children of a given node, select all nodes with greater L
values and lesser R values [3]. To determine the root node, select that node with an L value of 1. Lastly, all leaf nodes can be discerned by selecting the nodes with L values equaling their R values minus 1 [5].

Dynamic Interval Encoding is an enhancement to Interval Encoding for supporting XQuery FLWR expressions. In the absence of special indices on XML tuples, strategies for implementing nested FLWR expressions, even in native XML systems, have yielded at best quadratic performance in querying. Dynamic Interval Encoding achieves linear performance for nested FLWR expressions by collapsing FLWR expressions to a single SQL query. Basic SQL templates are defined for each function necessary to fully support FLWR expressions. These templates can be combined and nested as needed by supplying forest (or environment) widths during query execution [3]. The Dynamic Interval enhancements do not require any additional data beyond that which is required by Interval Encoding to be stored. Rather, it generates indices and views dynamically based on input widths.

2.2.3 ORDPATH Encoding

ORDPATH Encoding is essentially an enhanced version of Dewey Order Encoding that uses indefinite path lengths to eliminate the need for renumbering nodes on insertions [4]. To reduce the rate at which paths increase, ORDPATH incorporates three additional changes into the Dewey strategy for assigning control information to object elements:

- New objects are all initially assigned odd-numbered indices when inserted into a database. This leaves even numbers open for later inserts.
- Negative indices are supported as a way of inserting a sibling to the left of the first sibling of a node: in document terms, inserting a child before or above all other children of an existing node.
- Identifiers (both element names and attribute names) are assigned a unique integer tag stored in the main table with a second tag-to-identifier mapping table for recovering the original identifiers.

An example of an XML document and its ORDPATH encoding are given in Figure 2-3 and Listing 2-3.

The contrast between Figures 2-2 and 2-3 illustrates key differences between ORDPATH and Interval Encoding. First, the ORDPATH encoding in Figure 2-3, unlike Interval encoding, assigns a
unique tag to each identifier in the XML document. The encoding of identifiers as tag numbers saves a significant amount of space when encoding XML documents with many repeated identifiers, which should be the case for data- and document-centric documents. For documents that are neither data- nor document-centric, the benefit of encoding identifiers as integers is less certain. Second, the node type (element, attribute, comment or value) is encoded in a separate column, once again as an integer (a single byte). This node type field provides direct support for XML queries that select node by type; in Interval Encoding, as described by DeHaan, type queries are supported using stored procedures to do string parsing and comparison [4].

### Listing 2-3: Another XML Fragment [4]

```xml
<book isbn="1-55860-438-3">
  <section>
    <title>Bad Bugs</title>
    Nobody loves bad bugs.
    <figure caption="Sample bug"/>
  </section>
  <section>
    <title>Tree Frogs</title>
    All right-thinking people love tree frogs.
  </section>
</book>
```

### Figure 2-3: ORDPATH Encoding of Listing 2-3 [4]

<table>
<thead>
<tr>
<th>ORDPATH</th>
<th>Tag</th>
<th>Node Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1 (book)</td>
<td>1 (Element)</td>
<td>NULL</td>
</tr>
<tr>
<td>1.1</td>
<td>2 (isbn)</td>
<td>2 (Attribute)</td>
<td>'1-55860-438-3'</td>
</tr>
<tr>
<td>1.3</td>
<td>3 (section)</td>
<td>1 (Element)</td>
<td>NULL</td>
</tr>
<tr>
<td>1.3.1</td>
<td>4 (title)</td>
<td>1 (Element)</td>
<td>'Bad Bugs'</td>
</tr>
<tr>
<td>1.3.3</td>
<td>NULL</td>
<td>4 (Value)</td>
<td>'Nobody loves bad bugs.'</td>
</tr>
<tr>
<td>1.3.5</td>
<td>5 (figure)</td>
<td>1 (Element)</td>
<td>NULL</td>
</tr>
<tr>
<td>1.3.5.1</td>
<td>6 (caption)</td>
<td>2 (Attribute)</td>
<td>'Sample bug'</td>
</tr>
<tr>
<td>1.5</td>
<td>3 (section)</td>
<td>1 (Element)</td>
<td>NULL</td>
</tr>
<tr>
<td>1.5.1</td>
<td>4 (title)</td>
<td>1 (Element)</td>
<td>'Tree frogs'</td>
</tr>
<tr>
<td>1.5.3</td>
<td>NULL</td>
<td>4 (Value)</td>
<td>'All right-thinking people'</td>
</tr>
<tr>
<td>1.5.5</td>
<td>7 (bold)</td>
<td>1 (Element)</td>
<td>'love'</td>
</tr>
<tr>
<td>1.5.7</td>
<td>NULL</td>
<td>4 (Value)</td>
<td>'tree frogs'</td>
</tr>
</tbody>
</table>
The ORDPATH itself is a binary encoding of a Dewey-like path that maintains document order and allows cheap and easy node comparisons. This encoding consists of a set of L_i/O_i bitstring pairs: one for each component of the ORDPATH. Each component’s L_i portion must satisfy these requirements:

- “Given that one knows where a L_i bitstring starts, he/she can identify where it stops” (i.e., each L_i bitstring must be unique and may not start with (scanning from left to right) the same bit pattern as any other L_i bitstring) [4].

- “Each L_i bitstring specifies the length in bits of the succeeding O_i bitstring [4].”

- The values chosen for L_i bitstrings must maintain document order (when bit-by-bit binary comparisons are done from left to right) [4].

The mapping from L_i bitstrings to O_i bitstring lengths and value ranges is stored in a separate table, as shown in Figure 2-4:

<table>
<thead>
<tr>
<th>L_i Bitstring</th>
<th>L_i</th>
<th>O_i value range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000001</td>
<td>20</td>
<td>[-1118485, -69910]</td>
</tr>
<tr>
<td>00000001</td>
<td>16</td>
<td>[-69909, -4374]</td>
</tr>
<tr>
<td>0000001</td>
<td>12</td>
<td>[-4373, -278]</td>
</tr>
<tr>
<td>000001</td>
<td>8</td>
<td>[-277, -22]</td>
</tr>
<tr>
<td>00001</td>
<td>4</td>
<td>[-21, -6]</td>
</tr>
<tr>
<td>0001</td>
<td>2</td>
<td>[-5, -2]</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>[-1, 0]</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>[4, 7]</td>
</tr>
<tr>
<td>1110</td>
<td>4</td>
<td>[8, 23]</td>
</tr>
<tr>
<td>11110</td>
<td>8</td>
<td>[24, 279]</td>
</tr>
<tr>
<td>111110</td>
<td>12</td>
<td>[280, 4375]</td>
</tr>
<tr>
<td>1111110</td>
<td>16</td>
<td>[4376, 69911]</td>
</tr>
<tr>
<td>11111110</td>
<td>20</td>
<td>[69912, 1118487]</td>
</tr>
</tbody>
</table>

This table, for example, would be used to encode the ORDPATH ‘1.5.3.-9.11’ as the bitstring 011100110100001110011100011. This encoding is produced by locating each component value in the O_i value ranges and appending the corresponding L_i bitstring followed by the corresponding number of bits.
specifying the offset for the component value from the minimum $O_i$ value for that range. For the example, consider the following translations from component values to $L_i$ bitstrings + $O_i$ offsets:

1 \rightarrow 01 \text{ (offset of 0 from 1 specified in 0 bits)}
5 \rightarrow 110 + 01 \text{ (offset of 1 from 4 specified in 2 bits)}
3 \rightarrow 10 + 1 \text{ (offset of 1 from 2 specified in 1 bit)}
-9 \rightarrow 00001 + 1100 \text{ (offset of 12 from –21 specified in 4 bits)}
11 \rightarrow 1110 + 0011 \text{ (offset of 3 from 8 specified in 4 bits)}

A set of fairly straightforward algorithms supports the traversal of ORDPATH trees. For instance, to determine the parent of a given node, simply drop the last component of its ORDPATH. To determine siblings, simply increment the last component of the ORDPATH.

The one clear algorithmic advantage of Interval Encoding over ORDPATH is that a node’s indices can be used to predict how many descendants that node has. There is no such shortcut with ORDPATH.

2.3 Summary

This chapter included background information on XML and XQuery. It described three basic techniques for shredding XML documents—Global, Local and Dewey Order Encodings—and two alternative encoding strategies for enhancing their performance: Interval Encoding, a variation of Global Order Encoding that essentially adds to a node’s label the width of the forest rooted by the node, and ORDPATH, a variation of Dewey Order Encoding that specifies a procedure for expressing a path as a binary bitstring.
CHAPTER 3
EXPERIMENTAL METHODS

3.1 Motivation & High-Level Strategy

As stated previously, this research sought to investigate methods of encoding XML documents in relational databases. This is an important topic because of XML’s growing popularity and RDBMS market penetration. Major relational database vendors (Microsoft, Oracle and IBM) now support XML with a layer on top of their RDBMS. Since there were no published studies comparing ORDPATH to Interval Encoding, the decision was made to implement and compare both approaches, determining especially what circumstances the insert-friendly approach of ORDPATH would pay off.

Work on this research began with extensive background research on XML, XQuery, and some related but ultimately irrelevant topics (including Grammars, Compilers, and Finite State Machines). The topic was an extension of a project for a Database Design course on document shredding.

The work described here compared ORDPATH and Interval Encoding by determining the relative performance of insertion and selection operations for the two encodings, relative to the 20 queries that make up the XMark⁵ benchmark suite. These actions, which were specifically designed to test performance of XML databases, were each applied to an XML file representing the data for a mock auction (such as would be needed for a site such as Ebay). Listing 3-1 exemplifies the structure of these XML files:

---

Listing 3-1: Example auction.xml

```xml
<site>
  <regions>
    <africa>
      <item id="item0">
        <location>United States</location>
        <quantity>1</quantity>
        <name>duteous nine eighteen </name>
        <payment>Creditcard</payment>
        <decription> ... </decription>
        ...
      </item>
    </africa>
  </regions>
</site>
```

---

⁵ see [http://monetdb.cwi.nl/xml](http://monetdb.cwi.nl/xml) (retrieved on 15 Mar 2006)
Listing 3-1 (Continued)

```xml
...<africa>
...<regions>
<categories>
<category id="category0">
  <name>liquor</name>
  <description>
    <text>
      fondness vines consenting censuring preventions
    </text>
  </description>
</category>
...<categories>
<people>
<person id="person0">
  <name>Klemens Pelz</name>
  <emailaddress>mailto:Pelz@duke.edu</emailaddress>
  <creditcard>6491 3985 6149 1938</creditcard>
  <watches>
    <watch open_auction="open_auction23"/>
  </watches>
</person>
...<people>
<open_auctions>
<open_auction id="open_auction0">
  <initial>70.44</initial>
  <reserve>391.57</reserve>
  <bidder>
    <date>12/02/2001</date>
    <time>05:07:46</time>
    <personref person="person175"/>
    <increase>9.00</increase>
  </bidder>
</open_auction>
...<open_auctions>
<closed_auction>
  <seller person="person166"/>
  <buyer person="person148"/>
  <itemref item="item4"/>
  <price>18.00</price>
  <date>03/22/1999</date>
  <quantity>1</quantity>
  <type>Featured</type>
  <annotation>
    <description>
      entering mock means sore orders allay friendly
    </description>
    <happiness>6</happiness>
  </annotation>
</closed_auction>
...<closed_auctions>
</site>
```
The following is a brief, intuitive description of each query\(^6\):

- Query 1 lists the names of all people stored in the data file.
- Query 2 returns the bid increase amount for the second bidder in each open auction.
- Query 3 returns the first and last increase amounts for all open auctions where the last increase is greater than 2 times the first increase.
- Query 4 returns the reserve amount for any open auctions where person18829 bidded before person10487.
- Query 5 returns the number of closed auctions where the price is greater than 40.
- Query 6 returns the number of \(<\text{item}\>\) elements in all regions (regardless of generation or tree depth).
- Query 7 returns the number of \(<\text{annotation}\>\)’s plus the number of \(<\text{emails}\>\)’s plus the number of \(<\text{description}\>\)’s in the site (regardless of generation or tree depth).
- Query 8 returns each person and the number of closed auctions they won.
- Query 9 returns each person and the names of the items that originate from Europe that they bought.
- Query 10 returns each interest category and the people who have that interest transformed as French text.
- Query 11 returns each person and the number of open auctions their income exceeds by a scale factor of 5000.
- Query 12 returns the incomes of the persons and the count of open auctions their income exceeds by a scale factor of 5000, but only if their income is greater than 50000.
- Query 13 returns the names and descriptions of the items originating in Australia.
- Query 14 returns the name of the items in the site whose descriptions contain the word ‘Gold’ (without regard to tree depth).
- Query 15 returns the keywords that are nested two levels deep in lists contained in the description of a closed auction’s annotation.
- Query 16 returns the sellers of items that have non-empty keywords as described in Query 15.
- Query 17 returns the names of persons whose homepage is an empty string.
- Query 18 returns the reserves of all open auctions converted to Euros.
- Query 19 returns the names and locations of all \(<\text{item}\>\)’s in all regions (regardless of tree depth).

\(^6\) See Appendix A.5 for query implementations
• Query 20 returns 4 groupings of customer’s based on income: ‘preferred’, ‘standard’, ‘challenge’, and ‘na’.

Test XML files of sizes 1.5 MB, 3.0 MB and 4.5 MB, corresponding to XMark scale factors of 0.01, 0.02 and 0.03 respectively, were used. Database operations were executed on a single PC with one processor, which ran the MySQL server and the Python client code as concurrent processes. The PC was rebooted before any benchmarks were run and all non-essential processes were killed. Each benchmark was run once to fill the MySQL query cache and then run 5 subsequent times. There was only a slight difference in running times of the first and second runnings for some of the benchmarks—others were unchanged. MySQL does not have a data cache but, of course, the operating system caches pages. During query execution, a momentary disk read occurred approximately once every 3 seconds for Interval Encoding and once every 6 seconds for ORDPATH Encoding. Additional caching would probably have had little affect on the results—though future investigations of database caching would be of interest. During insert execution, disk activity was constant for Interval and intermittent for ORDPATH. MySQL uses B-Trees for its indices and they are updated on each insert.

Performance comparisons were based on wall-clock time. The Python function ‘time.clock()’ was used to for all measurements. CPU time, such as returned by the Python benchmarking utility HotShot, was inappropriate for measuring performance: CPU time data excluded time spent in the database and for network transfer which unfairly favored Interval time, whose algorithms require much less local CPU use. This difference in CPU time can be mainly attributed to ORDPATH’s encoding of paths to and from bitstrings (which is CPU-intensive). The queries, the encoding methods, and the query-supporting libraries were implemented in the Python programming language. Python was selected for its rapid development capabilities and platform independence.

After intial results were in, it became clear that running a mix of selects and inserts may produce different results than the uniform running pattern used in benchmarking each query individually and, in an effort to validate the conclusions drawn from the results of the benchmarks, 5 different mixes of 100 operations (selects/inserts) were run for each file size—88/12, 85/15, 82/18, 79/21, and 76/24.
The query-supporting libraries adhered to a common interface, which allowed each query to be coded once. Listing 3-2 gives this interface’s specification\(^7\): i.e., the interface definition for a base class that supports XML document traversal—*abstractTraverser*. The *abstractTraverser* class was subclassed to obtain derived classes that implement the Interval and ORDPATH traversal algorithms. Where possible, *abstractTraverser*’s methods were defined in terms of polymorphic (i.e., ORDPATH or Interval) realizations of other *abstractTraverser* methods. However, in most cases, a more efficient implementation of these methods could be defined in the derived classes proper.

\(^7\) See Appendix A for complete listings of all source codes produced for this project.
3.2 Low-Level Strategy & Implementation Details

This section contains an example query along with its implementation and definitions of key supporting functions for both Interval and ORDPATH encodings. Every query in the XMark suite was encoded by hand, using primitives from abstractTraverser. XMark Query 3, shown in the header comment of Listing 3-4, typifies how queries were implemented.

- The final loop in q3, which iterates through all auctions returned by the call to getChildrenByName, corresponds to Query 3’s main ‘for’ loop.
- The first eight lines of processAuction, which retrieve the first bidder, his/her increase, the last bidder, and his/her increase, correspond to the expressions in Query 3’s WHERE clause. If any of those accesses fail, the code returns immediately.
- The final ‘if eval…’ statement in processAuction completes the processing for Query 3’s WHERE clause. If the conditions are met, processAuction prints the result to standard output (which corresponds to the RETURN clause of the FLWR statement).

Originally, each query used getNextSibling to iterate through a node’s children. The getNextSibling function, however, was replaced by Interval-encoding-specific and ORDPATH-encoding-specific (cf. Listing 3-5 & Listing 3-3) implementations of getChildrenByName after tree-traversal proved inefficient for Interval and ORDPATH.

Listing 3-3: Interval-Specific Implementation of getChildrenByName

```python
def getChildrenByName( self, node, name ):
    self.cursor.execute( """SELECT * FROM %s WHERE l > %s AND l < %s AND
    nodeType = %s AND depth = %s AND s = '%s';" " % \
    ( self.tablename, node[self.L_INDEX], node[self.R_INDEX], \ 
    self.ELEMENT_NODE_TYPE, node[self.DEPTH_INDEX] + 1, name ))
    return self.cursor.fetchall()
```

28
For the Interval-encoding-specific code (Listing 3-3), a node’s children match the following conditions:

```python
#!/usr/bin/env python
#
# --------------------------------------------------------------
# Query 3
# --------------------------------------------------------------
# FOR    $b IN document("auction.xml")/site/open_auctions/open_auction
# WHERE  $b/bidder[0]/increase/text() *2 <=
# $b/bidder[last()]/increase/text() 
# RETURN <increase first=$b/bidder[0]/increase/text()
#                  last=$b/bidder[last()]/increase/text()/>
import sys, os
sys.path.append( '../shared' )
from util import *
if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q3' )
    def processAuction( auction ):
        firstBidder = traverser.getFirstChildByName( auction, 'bidder' )
        if not firstBidder: return
        firstIncrease = traverser.getFirstChildByName( firstBidder, 'increase' )
        if not firstIncrease: return
        lastBidder = traverser.getLastChildByName( auction, 'bidder' )
        if not lastBidder: return
        lastIncrease = traverser.getFirstChildByName( lastBidder, 'increase' )
        if not lastIncrease: return
        firstIncrease = traverser.getNodeText( firstIncrease )[0]
        lastIncrease = traverser.getNodeText( lastIncrease )[0]
        if eval( firstIncrease ) * 2 <= eval( lastIncrease ):
            print '<increase first=%s
last=%s />' % ( firstIncrease, lastIncrease )
    def q3():
        rootNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        if not rootNode: return
        auctions = traverser.getChildrenByName( rootNode, 'open_auction' )
        for auction in auctions:
            processAuction( auction )
    doProfileAndPrintStats( q3, None )
cursor.close()
conn.close()
```
• L is less than the current node’s R value and greater than the current node’s L value: i.e., $l > node[L_INDEX]$ AND $l < node[R_INDEX]$.

• Nodetype is an element node: i.e., $nodeType = self.ELEMENT_NODE_TYPE$.

• Depth is one greater than current node: i.e., $depth = node[self.DEPTH_INDEX] + 1$.

• S is equal to the search string (name): i.e., $s = '%s' % name$

The key to this SQL statement is the name-matching clause. This clause essentially flattens the part of the database matching the other search criteria and uses the built-in index of node text values to find the proper matches. Additional indices on the nodeType and depth columns also speed the query. A high-quality SQL implementation would use optimizers to process the condition that returns the smallest set first, thereby speeding the remaining conditional matches and overall query execution. One could theoretically implement an entirely separate optimization layer on top of the SQL engine that utilizes ‘SELECT COUNT’ statements and other metadata to dynamically construct the SQL statements the engine will process. Although implementing this optimization system is beyond the scope of the current thesis, it is the basis, along with other claims made by designers of XML-enabled systems (such as DeHaan’s Dynamic Interval Encoding) for the belief that an XML-enabled system could potentially match or outperform native XML systems.

---

**Listing 3-5: ORDPATH-Specific Implementation of getChildrenByName**

```python
def getChildrenByName( self, node, name ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    tag = self.getTagFromIdentifier( name )
    if not tag: return None
    self.cursor.execute( """SELECT * FROM %s
        WHERE ordpath > '%s' AND ordpath < '%s' AND
        depth = %s AND tag = %s;"""
        % ( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
            upperBound, node[self.DEPTH_INDEX] + 1, tag ) )
    res = self.cursor.fetchall()
    node[self.ORDPATH_INDEX].incrementLastComponent( -1 )# reset original
    node ordPath object
    return res
```
For the ORDPATH-encoding-specific code (Listing 3-5) the conditions that a node’s children match are as follows:

- Ordpath is greater than the current node’s and less than the upper bound (which is defined as the current ordpath with the last component incremented): i.e., \( \text{ordpath} > '%s' \text{ AND ordpath} < '%s' \) (node[self.HEXSTRING_PATH_INDEX], upperBound).

Notice that Ordpath uses a string comparison, instead of the integer comparisons for Interval.

- Depth is one greater than current node: i.e., \( \text{depth} = \text{node}[\text{self.DEPTH_INDEX}] + 1 \).

- Tag is equal to the tag for the search string (name): i.e., \( \text{tag} = '%s' \text{ % tag} \). The tag value is found by the call to getTagFromIdentifier, which locates the value in a separate tag-to-value mapping table.

These conditions resemble those for Interval Encoding: nodeType, the only one that differs, could be omitted because restrictions on tag and depth eliminate all node types but element. Caching of Python Ordpath objects speeds up queries. The function getOrCreateOrdpath takes a node returned from the database and either returns an Ordpath object if it has already been created or creates it (by interpreting the meaning of the hexstring representation of the ORDPATH) and appends it to the node (which is a Python list). Caching allows a node to be used in the calling code with no additional performance penalty after it is first created.

The implementations of the ORDPATH and Interval algorithms were made more efficient by adding a depth column to the relational representation of documents. In the case of Interval Encoding, the depth column allows query-processing logic to discard nodes that appear to satisfy query conditions, but are in fact descendants that are not of interest (e.g., when searching for children of a given node with certain conditions). This depth information could only be computed by examining the label structure of descendant nodes from a node with a certain known depth. For ORDPATH-encoded data, the depth column eliminates the need for an expensive procedure to compute a node’s depth from its encoding. In the absence of a depth column, query implementations that used repeated calls to getNextSibling
outperformed implementations that used getChildrenByName. This use of the depth column slowed some queries, but yielded the best overall performance: an observation that suggested the need for an optimization layer that attaches conditions to queries dynamically, based on metadata such as the number of results which would be returned by a certain query conditions. This information could be obtained by executing a less expensive ‘SELECT COUNT’ statement instead of the actual query and used to order the conditions in much the same way the SQL engine does internally. Other metadata, of course, could be used for this optimization layer as well (such as which columns have indices and which do not and the types of and frequency of data).
CHAPTER 4
DATA

All benchmarks were run five times on a Celeron 2.2 GHz processor running Windows XP Home Edition
with 512 MB of RAM utilizing the MySQL\textsuperscript{8} 5.0.4-beta-nt database. Each benchmark was run one time to
fill the MySQL query cache. Then the subsequent 5 runs were averaged and recorded in Figure 4-1.

Figure 4-2 shows the database and index sizes for each encoding method. Figure 4-3 shows the running
times for heterogeneous, randomly chosen mixes of selects (first figure) and inserts (second figure).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
 & \multicolumn{3}{|c|}{Interval} & \multicolumn{3}{|c|}{Ordpath} \\
\hline
XMark scale & 0.01 & 0.02 & 0.03 & 0.01 & 0.02 & 0.03 \\
\hline
Q1: & 0.03389062 & 0.050767695 & 0.483571795 & 0.084568569 & 0.100181797 & 7.537545135 \\
Q4: & 0.3954483 & 0.61222449 & 1.250773772 & 2.349757302 & 3.680001676 & 5.596107047 \\
Q5: & 0.259290217 & 0.503450508 & 1.868231856 & 0.058557603 & 0.066783348 & 7.390835986 \\
Q6: & 0.103836381 & 0.190173751 & 1.508761868 & 0.310830846 & 0.239832297 & 1.473427979 \\
Q7: & 0.304454058 & 0.559815074 & 4.420741184 & 0.139898786 & 0.464513451 & 2.797863136 \\
Q9: & 23.19878504 & 96.81281539 & 314.5581092 & 175.6532895 & 869.9219312 & 5441.627869 \\
Q13: & 0.300463327 & 0.612591125 & 1.508761868 & 0.310830846 & 0.239832297 & 1.473427979 \\
Q17: & 1.176946511 & 2.042214164 & 3.40133689 & 6.864609329 & 11.67782023 & 17.82793623 \\
Q18: & 0.451598737 & 0.802419721 & 1.333466404 & 2.244224336 & 3.757220708 & 5.756101963 \\
Query average: & 3.349637418 & 9.848252287 & 46.34926272 & 18.32202442 & 59.9646773 & 301.7990454 \\
Insert 70: & 1333.173308 & 2606.242325 & 4199.181215 & 26.69041624 & 40.26645752 & 208.9745623 \\
Insert 100: & 2223.84736 & 4549.038209 & 6416.56104 & 38.5838 & 58.56196053 & 309.4376 \\
Insert average: & 1106.463556 & 2261.795333 & 3402.777372 & 20.78859238 & 32.01416225 & 164.5300362 \\
\hline
\end{tabular}
\end{table}

\textsuperscript{8} See \url{http://www.mysql.com/} (retrieved on 15 Mar 2006)
Figure 4-2: Encoded File Sizes (in bytes)

<table>
<thead>
<tr>
<th>Interval Ordpath</th>
<th>XMark scale</th>
<th>0.01</th>
<th>0.02</th>
<th>0.03</th>
<th>0.01</th>
<th>0.02</th>
<th>0.03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datafile size:</td>
<td>0.01</td>
<td>2,563,036</td>
<td>5,114,860</td>
<td>7,696,116</td>
<td>1,748,432</td>
<td>3,523,292</td>
<td>5,310,000</td>
</tr>
<tr>
<td>Index size:</td>
<td>0.01</td>
<td>2,322,432</td>
<td>4,561,920</td>
<td>6,910,976</td>
<td>1,341,440</td>
<td>2,581,504</td>
<td>3,943,424</td>
</tr>
</tbody>
</table>

Figure 4-3: Running Times for Heterogeneous Mixes of Benchmarks

<table>
<thead>
<tr>
<th>Interval Ordpath</th>
<th>XMark scale</th>
<th>0.01</th>
<th>0.02</th>
<th>0.03</th>
<th>0.01</th>
<th>0.02</th>
<th>0.03</th>
</tr>
</thead>
<tbody>
<tr>
<td>88/12 mix:</td>
<td>0.01</td>
<td>633.3603762</td>
<td>8615.328945</td>
<td>12197.74227</td>
<td>2389.689766</td>
<td>13979.85228</td>
<td>35239.04713</td>
</tr>
<tr>
<td>85/15 mix:</td>
<td>0.01</td>
<td>507.4515172</td>
<td>6627.70985</td>
<td>11734.12373</td>
<td>1874.216816</td>
<td>12924.17084</td>
<td>29376.46781</td>
</tr>
<tr>
<td>82/18 mix:</td>
<td>0.01</td>
<td>570.547975</td>
<td>11979.90577</td>
<td>17787.11346</td>
<td>2213.795138</td>
<td>17356.16804</td>
<td>72263.38954</td>
</tr>
<tr>
<td>79/21 mix:</td>
<td>0.01</td>
<td>609.8291414</td>
<td>7825.157828</td>
<td>21126.1911</td>
<td>1793.117581</td>
<td>11415.80042</td>
<td>51248.85994</td>
</tr>
<tr>
<td>76/24 mix:</td>
<td>0.01</td>
<td>676.678463</td>
<td>5268.479674</td>
<td>17896.35238</td>
<td>1859.414079</td>
<td>11619.01992</td>
<td>32111.42076</td>
</tr>
</tbody>
</table>

Figure 4-4: Interval Query Performance
Figure 4-5: ORDPATH Query Performance

Figure 4-6: Query Comparison – XMark 0.01 Document
Figure 4-9: Query Comparison – Average

Figure 4-10: Interval Insert Performance
Figure 4-11: ORDPATH Insert Performance

Figure 4-12: Insert Comparison – XMark 0.01 Document
Figure 4-13: Insert Comparison – XMark 0.02 Document

Figure 4-14: Insert Comparison – XMark 0.03 Document
Figure 4-15: Insert Comparison – Average

Figure 4-16: Heterogeneous DB Operation Comparison
Figure 4-17: Encoding Size Comparison – XMark 0.01 Document

Figure 4-18: Encoding Size Comparison – XMark 0.02 Document
Figure 4-19: Encoding Size Comparison – XMark 0.03 Document
CHAPTER 5

ANALYSIS

5.1 Qualitative Analysis

Of the three Order-Encoding Methods, Local is the least useful, and offers no advantage aside from space-efficiency over ORDPATH. Accordingly, the more interesting comparison is between the Global (or Interval) and Dewey (or ORDPATH) strategies for document encoding. Global is ideal for situations where a query-mostly workload is expected and Dewey is best for a mixture of queries and updates. These same observations apply to Interval and ORDPATH Encodings in their basic form. Interval Encoding, however, can be enhanced to provide better support for insertions. These enhanced encodings, L-Trees and Nested Interval Encoding [6] [7], are beyond the scope of this research, but their support for insertions is apparently comparable to that of ORDPATH (especially Nested Interval Encoding).

It is difficult to assess whether ORDPATH or Interval Encoding would tend to produce smaller databases. The overhead introduced by each method is sensitive to data in use. ORDPATH’s use of a ‘Tag’ column seems to be a win in this respect unless the document being shredded has very few reused tags and/or is mostly NULLs, in the (unlikely but possible) case of documents that consist primarily of content (instead of elements and attributes). ORDPATH’s ‘Node Type’ column requires more space, but it also improves the performance of type-related queries. Finally, the use of NULL to designate the type of complex elements, which would be common for data-centric XML documents, could waste substantial space. Once again, however, this use of space should enhance query efficiency.

5.2 Quantitative Analysis

5.2.1 Shred Comparison

Shredding times were dramatically different for Interval and ORDPATH. This can be attributed to the fact that SAX was used in slightly different ways. ORDPATH used a TreeWalker to do its parsing because a sequential scan as performed by ContentHandler, which Interval used, was insufficient.
5.2.2 Query Performance Comparison

As per the data in Chapter 4, Interval had an average of 3.35 seconds for the XMark 0.01 queries, 9.85 seconds for the 0.02 queries, and 46.35 seconds for the 0.03 queries. As Figure 4-4 shows, queries 3, 9 and 10 were largely responsible for the difference. These queries are scales of magnitude more complex than the others due to their nested-loop structure. The XMark 0.02 file is only twice as large as the XMark 0.01 file; the average query time, however, was almost 3 times slower. The XMark 0.03 file is only 3 times larger than the XMark 0.01 file; however, the average query time was 14 times slower (see Figure 4-9). This can be attributed to the nested traversal-based nature of the queries (esp. queries 3, 9 and 10). Further support for this claim can be obtained by considering any of the queries that do not have a nested-loop structure. Take, for instance, query 2; the times for Interval encoding for XMark 0.01, 0.02 and 0.03 are 2.92s, 5.84s and 9.34s respectively, which is approximately a scaling of 2 and 3.

Figures 4-1 and 4-5 show a similar pattern for ORDPATH encoding. For instance, again take query 2; the times for ORDPATH encoding for XMark 0.01, 0.02 and 0.03 are 3.89s, 7.35s and 11.94s respectively, which is also a scaling of approximately 2 and 3. Also, the times for more complex queries such as query 9 are scaling comparably to Interval. However, Interval outperforms ORDPATH on select queries by an average of 5.47 times for XMark 0.01, 6.09 times for XMark 0.02, and 6.51 times for XMark 0.03. Overall, Interval outperforms ORDPATH by 6.38 times. This figure was calculated by taking the sum of the averages for both approaches and dividing (380.09s / 59.55s). If one extrapolates this out to larger file sizes, it would appear that Interval would continue to outperform ORDPATH by significantly increasing margins (see Figure 4-9). If queries 3, 9, and 10 were thrown out, the data would still favor Interval, but by a moderately smaller margin.

5.2.3 Insert Performance Comparison

The data in Figure 4-1 shows that ORDPATH, as expected, greatly outperforms Interval Encoding for inserts. When using amortized analysis, Interval turns out to have a running time on the order of O(c) for
a given document size although it has to renumber, on average, half the nodes. So, both algorithms are essentially $O(C)$ for inserts (for a given document size) although that $C$ is much greater for Interval.

The data in Figure 4-10 shows a roughly symmetric spread-out for Interval inserts as document sizes and number of inserts increase. However, Figure 4-11 shows an asymmetrical spread-out for ORDPATH—inserts for the largest document size follow a much steeper slope than for the smaller documents. This, the author suspects, is due to the lengthening of the ORDPATH itself as the number of nodes in the document increase. Since the ORDPATH field is the primary key for the ORDPATH encoding, it is slowing both queries and inserts (see Figures 4-9 and 4-11) possibly due to the growing size of the primary key index.

For the XMark 0.01 document, ORDPATH outperforms Interval by a factor of 53.22; for the 0.02 document, by a factor of 70.65; and for the 0.03 document, by a factor of 20.68. Thus, the margin is decreasing as document sizes increase. This is also most likely due to the lengthening of the primary key for the ORDPATH encoding and the resulting increase in index size and the increased use of CPU for translating the longer paths to and from bitstrings.

Overall, ORDPATH outperforms Interval for inserts by a factor of 31.16 times. This figure was calculated by taking the sum of the average insert times for the 0.01, 0.02 and 0.03 documents for both approaches and dividing $(6771.04s / 217.33s)$. This yields the equation:

$$30.16 \times \text{numInserts} - 5.38 \times \text{numSelects} = 0$$

which describes the relation between the number of inserts and selects to achieve the perfect balance where both methods would perform equally well. One can use this equation to see if the ORDPATH approach is beneficial (for document sizes in the range tested). For example, consider a situation where a database gets 10 insert queries per hour. Here, one would need to have 56 ($\frac{(30.16 \times 10)}{5.38}$) select queries per hour to achieve the balance between the methods, and greater than 56 select queries to start seeing any net performance gain. So, the equilibrium ratio of inserts to selects for these two methods is approximately 1:5.6.
5.2.4 Heterogeneous Mix Comparison

According to the data in Figures 4-3 and 4-16, Interval outperformed ORDPATH for every mix of selects and inserts and for each file size. This is a rather surprising result given the analysis contained in sections 5.2.1-5.2.3. This is most likely due to the fact that a standard PC was used to run these tests with a limited amount of memory. Interval encoding involved a much greater amount of disk activity for the homogeneous tests, but during the heterogeneous tests, both Interval and ORDPATH involved near constant disk access. Running these tests on a production server would likely produce different and possibly more relevant results.

5.2.5 Encoding Size Comparison

According to the data presented in Figures 4-2, 4-17, 4-18 and 4-19, there is a considerable difference in the encoding sizes of Interval and ORDPATH encodings. The increase in data and index file sizes from 0.01 to 0.02 and 0.03 is consistent with the scale factors of 2 and 3, which the original document sizes adhere to. Overall, Interval data and index sizes are 1.58 times larger than ORDPATH sizes (29,169,340 bytes/ 18,448,092 bytes). Thus, Interval opts for a space-for-time tradeoff that helps it to outperform on queries.
CHAPTER 6
CONCLUSION

6.1 Final Conclusions

Each of the two methods considered has its particular weaknesses and strengths. According to the homogeneous tests, if less than 18% of an application’s database operations (excluding updates which have no bearing on this research) are inserts, then it would not be beneficial to implement ORDPATH. However, the heterogeneous tests indicate that Interval will outperform ORDPATH up to a mix of 24% inserts and likely quite higher. One should have a good understanding of the type and form of the XML data to be shredded and its likely update pattern in order to piece together an implementation that works best for a given situation. Hybrids and combinations of the methods researched are possible and should be considered viable solutions.

6.2 Future Work

Future work in this area could include an XQuery implementation that utilizes the query-supporting libraries produced by this work. Enhancing the Interval Encoding with an insert-friendly approach, such as L-Tree, and doing more comparison between the encodings would also be of interest. It might also be beneficial to translate the Python code presented to a more efficient language such as C++ if the current performance is not satisfactory. Future work could also include running the benchmarks on other RDBMS such as Oracle or MS SQL Server. Additionally, tweaking database caching facilities in order to draw conclusions about caching effects on performance of the encoding approaches studied would also be of interest. It would also be interesting to see these tests run on a production-grade server with enough main memory to store the entire database. Finally, results for additional select/insert mixes between 75/25 and 25/75 would prove valuable.
WORKS CITED


# Appendix A

Complete Source Code Listings

A.1 Database Setup and Utility Functions

```python
# -------------------------------------------------------------
# setup.py
# --------
# Sets up or resets the databases and metatables for
# xml-rdbms (intrval and ordpath).
# -------------------------------------------------------------

from getpass import *
import MySQLdb
import sys, errno, os

if __name__ == '__main__':
    def printUsage( errMsg = None):
        if errMsg != None:
            print 'Error: %s' % errMsg
        print "Usage: setup.py mysqlUsername"
        sys.exit(1)

    if len( sys.argv ) != 2:
        printUsage()

    host = "127.0.0.1"
    user = sys.argv[1]
    passwd = getpass( 'Enter password: ' )
    port = 3306

    try:
        conn = MySQLdb.connect( host = host, user = user, passwd = passwd,
                                port = port )
    except MySQLdb.Error, e:
        print "Error %d: %s" % (e.args[0], e.args[1])
        sys.exit(1)

    cursor = conn.cursor()
    cursor.execute( 'DROP DATABASE IF EXISTS intrval;' )
    cursor.execute( 'CREATE DATABASE intrval;' )
    cursor.execute( 'DROP DATABASE IF EXISTS ordpath;' )
    cursor.execute( 'CREATE DATABASE ordpath;' )

    fh = open( '../ordpath/createMetatables.sql' )
    lines = fh.readlines()

    for line in lines:
        if line.strip():
            cursor.execute( line )

    cursor.execute( 'COMMIT;' )
    cursor.close()
    conn.close()
```

49
USE ordpath;
DROP TABLE IF EXISTS prefixMapping;
CREATE TABLE prefixMapping( prefix VARCHAR(10), Li TINYINT, Omin INT, Omax INT, PRIMARY KEY( prefix ));

INSERT INTO prefixMapping VALUES( '000000001', 20, -1118485, -69910 );
INSERT INTO prefixMapping VALUES( '00000001', 16, -69909, -4374 );
INSERT INTO prefixMapping VALUES( '0000001', 12, -4373, -278 );
INSERT INTO prefixMapping VALUES( '000001', 8, -277, -22 );
INSERT INTO prefixMapping VALUES( '00001', 4, -21, -6 );
INSERT INTO prefixMapping VALUES( '0001', 2, -5, -2 );
INSERT INTO prefixMapping VALUES( '001', 1, -1, 0 );
INSERT INTO prefixMapping VALUES( '01', 0, 1, 1 );
INSERT INTO prefixMapping VALUES( '10', 1, 2, 3 );
INSERT INTO prefixMapping VALUES( '110', 2, 4, 7 );
INSERT INTO prefixMapping VALUES( '1110', 4, 8, 23 );
INSERT INTO prefixMapping VALUES( '11110', 8, 24, 279 );
INSERT INTO prefixMapping VALUES( '111110', 12, 280, 4375 );
INSERT INTO prefixMapping VALUES( '1111110', 16, 4376, 69911 );
INSERT INTO prefixMapping VALUES( '11111110', 20, 69912, 1118487 );
import MySQLdb, sys
import hotshot, hotshot.stats
import time, os

DO_DETAILED_PROFILE = False

def getConnection():
    host = "127.0.0.1"
    user = "username"
    passwd = "password"
    port = 3306

    try:
        conn = MySQLdb.connect( host = host, user = user, passwd = passwd, port = port )
    except MySQLdb.Error, e:
        print "Error %d: %s" % (e.args[0], e.args[1])
        sys.exit(1)
    return conn

def doProfileAndPrintStats( func, *args ):
    if DO_DETAILED_PROFILE:
        it_profile = hotshot.Profile( 'func profile' )
        if args[0]:
            time.clock()
            it_profile.runcall( func, args[0] )
        else:
            time.clock()
            it_profile.runcall( func )
        print '%s took %s wall seconds.' % ( func, time.clock() )
        it_profile.close()
        stats = hotshot.stats.load( 'func profile' )
        stats.strip_dirs()
        stats.sort_stats( 'file', 'cumulative' )
        stats.print_stats()
        os.remove( 'func profile' )
    else:
        if args[0]:
            time.clock()
            func( args[0] )
        else:
            time.clock()
            func()
        print '%s took %s wall seconds.' % ( func, time.clock() )

def setupForQuery( scriptName, expectedArgCount = 3, variableNamesBeyondFirstThree = '' ):

def printUsage( errMsg = None ):
    if errMsg != None:
        print 'Error: %s' % errMsg
    print "Usage: %s.py -i|o tablename %s" % ( scriptName,
sys.exit(1)

if len(sys.argv) != expectedArgCount:
    printUsage()

tablename = sys.argv[2]

conn = getConnection()
cursor = conn.cursor()

if sys.argv[1] == '-i':
    from ilib import intervalTraverser
    cursor.execute('USE intrval;')
    traverser = intervalTraverser(cursor, tablename)
elif sys.argv[1] == '-o':
    from olib import ordpathTraverser
    cursor.execute('USE ordpath;')
    traverser = ordpathTraverser(cursor, tablename)
else:
    printUsage("'%(s should be -i (for interval) or -o (for ordpath)" % sys.argv[1])

return traverser, conn, cursor
import sys, os
import math, random
import popen2

sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
def printUsage():
    print 'Usage: mix.py tablename percentSelects'
sys.exit(0)

if len(sys.argv) != 3:
    printUsage()

intervalQueries = []
ordpathQueries = []

def addRandomSelect():
r = random.randint( 1, 20 )
intervalQueries.append( 'q%s.py -i %s' % ( r, sys.argv[1] ))
ordpathQueries.append( 'q%s.py -o %s' % ( r, sys.argv[1] ))

def mix():
    numSelects = eval(sys.argv[2])
    if numSelects < 50:
        print 'Number of selects must be greater than 50'
        printUsage()

    numSelectsPerInsert = math.floor( 
        numSelects / ( 100.0 - numSelects )).__int__()

    for i in range( 100 - numSelects ):
        intervalQueries.append( 'insert.py -i %s 1' % ( sys.argv[1] ))
        ordpathQueries.append( 'insert.py -o %s 1' % ( sys.argv[1] ))

    for j in range( numSelectsPerInsert ):
        addRandomSelect()

    for i in range( 100 - len( intervalQueries )):
        addRandomSelect()
def intervalMix():
    for q in intervalQueries:
        print 'Running: %s' % q
        os.popen( q )

def ordpathMix():
    for q in ordpathQueries:
        print 'Running: %s' % q
        os.popen( q )

mix()
doProfileAndPrintStats( intervalMix, None )
doProfileAndPrintStats( ordpathMix, None )
A.2 Shredders

A.2.1 Interval Shredder

```python
class contentHandler( saxutils.DefaultHandler ):
    ELEMENT_NODE_TYPE = 1
    ATTRIBUTE_NODE_TYPE = 2
    COMMENT_NODE_TYPE = 3
    VALUE_NODE_TYPE = 4
    insertedTotal = 0
    def __init__(self, conn, tablename):
        self.conn = conn
        self.cursor = self.conn.cursor()
        self.tablename = tablename
        self.curLVal = 0
        self.curDepth = 1
        self.p = re.compile( "'" ) # for replacing ' with '' in content
        self.createTimer()
        self.startDocument()

    def createTimer(self):
        self.t = threading.Timer( 7.0, self.printStatus )
        self.t.start()

    def printStatus(self):
        print 'total inserted: %s' % self.insertedTotal
        self.createTimer()

    def startDocument(self):
        self.cursor.execute( "USE intrval;" )
        self.cursor.execute( "DROP TABLE IF EXISTS " + self.tablename + ";;"
        self.cursor.execute( "CREATE TABLE " + self.tablename + \ 
            "( s TEXT, l INT, r INT, nodeType INT, depth INT, PRIMARY KEY( l ),
            INDEX( nodeType ), INDEX( depth ));" )
```
```python
def endDocument(self):
    self.cursor.execute( "COMMIT;" )
    self.cursor.close()
    self.conn.close()

def insertRecord( self, sval, lval, rval, nodeType, depth ):
    self.insertedTotal += 1
    self.cursor.execute("INSERT INTO %s VALUES('%s', %s, %s, %s, %s );" % 
        ( self.tablename, sval, lval, rval, nodeType, depth ))

def startElement(self, name, attrs):
    self.insertRecord( name, self.curLVal, 0, self.ELEMENT_NODE_TYPE, 
                        self.curDepth )

    self.curLVal += 1

    for k,v in zip( attrs.keys(), attrs.values() ):
        # insert k, l, l+3, ATTRIBUTE_NODE_TYPE
        self.insertRecord( k, self.curLVal, self.curLVal + 3,
                           self.ATTRIBUTE_NODE_TYPE, self.curDepth )

        self.curLVal += 1

        # insert v, l, l+1, VALUE_NODE_TYPE
        self.insertRecord( v, self.curLVal, self.curLVal + 1,
                           self.VALUE_NODE_TYPE, self.curDepth )

        self.curLVal += 1
        self.curLVal += 3
        self.curDepth += 1

def endElement(self, name):
    self.cursor.execute( "SELECT MAX(l) FROM " + self.tablename + " WHERE r = 0 AND nodeType = %s;" % self.ELEMENT_NODE_TYPE )
    lval = self.cursor.fetchone()[0]

    if lval == None:
        print "Error: couldn't find start element tag"
        sys.exit(1)

    self.cursor.execute( "UPDATE " + self.tablename + " SET r = " + 
                         str( self.curLVal ) + " WHERE l = " + str( lval ) + ";" )
    self.curLVal += 1
    self.curDepth -= 1

def characters(self, content):
    if content.strip():
        try:
            self.insertRecord( content, self.curLVal, self.curLVal + 1,
                               self.VALUE_NODE_TYPE, self.curDepth )
        except MySQLdb.Error, e:
            if e.args[0] == 1064:
                self.insertRecord( self.p.sub( '\\', content ),
                                   self.curLVal, self.curLVal + 1,
                                   self.VALUE_NODE_TYPE, self.curDepth )
            else:
                print "Error %d: %s" % (e.args[0], e.args[1])
        self.curLVal += 2
```

56
def error(self, e):
    print "Error: %r" % e
    sys.exit(1)

def fatalError(self, e):
    print "Fatal error: %r" % e
    sys.exit(1)

if __name__ == '__main__':
    import sys, errno, os

    if len(sys.argv) != 3:
        print "Usage: ishred.py filename tablename"
        sys.exit(1)

    try:
        f = file(sys.argv[1])
    except Exception, err:
        print "Error: cannot open file: " + sys.argv[1]
        sys.exit(1)

def doShred():
    conn = getConnection()

    parser = make_parser()
    parser.setFeature(feature_namespaces, 0)

    ch = contentHandler(conn, sys.argv[2])
    parser.setContentHandler(ch)
    parser.parse(f)

    doProfileAndPrintStats(doShred, None)
A.2.2 ORDPATH Shredder and Supporting Classes

# ------------------------------------------------------------
# ordpath.py
# -------
# Represents an ordpath. Provides encoding to/from bitstring
# and hexstring representations. Provides ORDPATH manipulation
# routines.
# ------------------------------------------------------------

from stringConverter import *
import copy

class ordPath(object):
    def __init__(self, cursor):
        self.list = []                # the ordpath as list
        self.string = ''              # the ordpath as bitstring
        self.cursor = cursor
        self.sc = stringConverter()

    def __str__(self):
        retVal = ''
        for i in self.list:
            retVal = retVal + '%d.' % i
        retVal = retVal[:-1]
        return retVal

    def __copy__(self):
        newCopy = ordPath( self.cursor )
        newCopy.list = copy.copy( self.list )
        newCopy.string = copy.copy( self.string )
        newCopy.sc = self.sc
        return newCopy

    def getListRepr(self):
        return self.list

    def getBitstringRepr(self):
        return self.string

    def getHexstringRepr(self):
        return self.sc.bitstring2hexstring( self.string )

    def setFromHexstring( self, hexstring ):
        self.string, self.list = self.decodeOrdpath( hexstring )
        hexstring = property( getHexstringRepr, setFromHexstring )

    def append(self, n):
        """appends new component with value of n""
        self.list.append(n)
        self.string = self.string + self.encodeOrdpathComponent(n)

    def dropLastComponent(self):
        """drops the last component of the ordpath (regardless of what
        components precede it)""
        assert( len(self.list) >= 1 )

58
self.string = \n    self.string[ :-len(self.encodeOrdpathComponent( self.list[-1] )))
self.list = self.list[:-1]

def dropLastComponentIncludeEvens(self):
    """drops all components from end of ordpath up
    to next to last odd component""
    assert( len(self.list) >= 1 )
    assert( self.list[-1] % 2 != 0 ) # last component must be odd
    assert( self.list[0] % 2 != 0 )  # first component must be odd
    i = 2
    l = copy.copy( self.list )
    self.dropLastComponent()
    while( l[-i] % 2 == 0 ):
        self.dropLastComponent()
        i += 1

def incrementLastComponent(self, n):
    """increments last component of ordpath by n""
    assert( self.list != [] )
    n = self.list[-1] + n
    self.dropLastComponent()
    self.append(n)

def encodeOrdpathComponent( self, n ):
    nstr = str(n)
    self.cursor.execute( """SELECT prefix, Li, Omin FROM prefixMapping
    WHERE Omin <= %s AND Omax >= %s;""" % \
    (nstr,nstr))

    res = self.cursor.fetchone()
    bitstring = res[0]

    Li = res[1]
    Omin = res[2]

    On = n - Omin
    On = '%x' % On
    On = self.sc.hexstring2bitstring( On ).lstrip( '0' )

    l = len(On)
    if 1 < Li:
        On = ( "0" * (Li - 1)) + On
    bitstring += On
    return bitstring

def encodeOrdpath( self, ordpath ):
    """returns encoded bitstring representation of ordpath.
    ordpath is a list of numbers [a, b, c, d] representing a.b.c.d""
    bitstring = ""
    for n in ordpath:
        bitstring += self.encodeOrdpathComponent( n )
    return bitstring
def decodeOrdpath( self, hexEncodedOrdpath ):
    if len( hexEncodedOrdpath ) < 1:
        raise ValueError('Error: hexstring must be at least one char long' )
    string = self.sc.hexstring2bitstring( hexEncodedOrdpath ).rstrip( '0' )
    list = []

    i = 0
    l = len( string )
    while( True ):
        j, comp = self.decodeOrdpathComponent( string[i:] )
        list.append( comp )
        i = i + j
        if i >= l:
            break
    return string, list

def decodeOrdpathComponent( self, bitstring ):
    firstBit = bitstring[0]
    i = 1
    l = len( bitstring )
    while( i < l and bitstring[i] == firstBit ):
        i += 1
    i = i + 1
    self.cursor.execute( """SELECT Li, Omin FROM prefixMapping WHERE prefix = '%s';""" % bitstring[:i] )
    res = self.cursor.fetchone()
    Li = res[0]
    Omin = res[1]
    if i + Li > l:
        raise ValueError( 'Malformed ordpath. Expecting another Oi component after %s' % Li )
    compVal = self.sc.bitstring2hexstringRtoL( bitstring[ i:Li+i ] )
    if compVal == '':
        compVal = '0'
    return Li + i, int( compVal, 16 ) + Omin

class tagDict(object): # TODO: make singleton ???
    def __init__(self):
        self.tagDict = {}
        self.curTag = 1

    def getTag( self, itemName ):
        if not self.tagDict.has_key( itemName ):
            self.tagDict[ itemName ] = str( self.curTag )
            self.curTag += 1
        return self.tagDict[ itemName ]

    def saveToDatabase( self, cursor, tablename ):
        for i in self.tagDict.items():
            cursor.execute('INSERT INTO ' + tablename + ' VALUES( %s, %s );' % i )
# stringConverter.py
# --------------------
# Converts hexstrings to bitstrings and vice versa.
# --------------------

class stringConverter( object ):
    def __init__(self):
        try:
            if stringConverter.bin2hex != None and \
                stringConverter.hex2bin != None: pass
        except:
            # initialize lookup tables
            stringConverter.bin2hex = {
                # Given a 4-char bitstring, return the corresponding 1-char
                # hexstring
                "0000": "0", "0001": "1", "0010": "2", "0011": "3",
                "0100": "4", "0101": "5", "0110": "6", "0111": "7",
                "1000": "8", "1001": "9", "1010": "a", "1011": "b",
                "1100": "c", "1101": "d", "1110": "e", "1111": "f",}
            stringConverter.hex2bin = {}
            for( bin, hex ) in stringConverter.bin2hex.items():
                stringConverter.hex2bin[ hex ] = bin

    def bitstring2hexstring( b ):
        ###"Take bitstring 'b' and return the corresponding l->r hexstring.""
        result = ""
        l = len(b)
        if 1 % 4:
            b = b + "0" * (4-(l%4))
        for i in range(0, len(b), 4):
            result = result + stringConverter.bin2hex[b[i:i+4]]
        return result
    bitstring2hexstring = staticmethod( bitstring2hexstring )

    def bitstring2hexstringRtoL( b ):
        ###"Take bitstring 'b' and return the corresponding r->l hexstring.""
        result = ""
        l = len(b)
        if 1 % 4:
            b = ("0" * (4-(l%4))) + b
        l = len(b)
        for j in [l - i for i in range(0, l, 4)]:
            result = stringConverter.bin2hex[b[j-4:j]] + result
        return result
    bitstring2hexstringRtoL = staticmethod( bitstring2hexstringRtoL )

    def hexstring2bitstring( h ):
        ###"Take hexstring 'h' and return the corresponding l->r bitstring.""
        result = ""
        for c in h:
            result = result + stringConverter.hex2bin[c]
        return result
    hexstring2bitstring = staticmethod( hexstring2bitstring )
import re
import copy
import sys, errno, os
import xml.dom
from xml.dom.ext.reader import Sax2
from xml.dom.NodeFilter import NodeFilter
from ordpath import *
sys.path.append( '../shared' )
from util import *

# Helper functions

def insertRecord( ordpath, tag, nodeType, value ):
    def insertRec( val):
        s = "INSERT INTO %s VALUES( '%s', %s, %s, '%s', %s );" % 
           (tablename, ordpath.getHexstringRepr(), tag, nodeType, val, curDepth)
        cursor.execute( s )
        value = value.strip()
        try:
            insertRec( value )
        except MySQLdb.Error, e:
            if e.args[0] == 1064:
                insertRec( p.sub( '\''', value ))
            else:
                print "Error %d: %s" % ( e.args[0], e.args[1] )

def processElement( name, attrs, content, ordpath, tagdict):
    if content == None:
        content = 'NULL'
    # insert ordpath, tag, 1 (Element), content
    insertRecord( ordpath, tagdict.getTag( name ), 1, content )
    if attrs:
        ordpath.append( -1 )
        for v in attrs.values():
            ordpath.incrementLastComponent( 2 )
            # insert ordpath, tag, 2 (Attribute), value
            insertRecord( ordpath, tagdict.getTag(v.nodeName), 2, v.nodeValue )

def processContent( content, ordpath ):
    insertRecord( ordpath, 'NULL', 4, content ) # 4 represents content
def processComment( comment, ordpath ):
    insertRecord( ordpath, 'NULL', 3, comment ) # 3 represents comment

def processNode( walker, ordpath, tagdict ):
global curDepth
curNode = walker.currentNode
attrs = curNode._get_attributes()
childIndices = filter.getAcceptedChildrenIndices( curNode )
numChildren = len( childIndices )

curDepth += 1
if checkComment and curNode.__class__ == xml.dom.Comment.Comment:
    assert( numChildren == 0 )
    assert( not attrs )
    processComment( curNode.nodeValue, ordpath )
elif curNode.__class__ == xml.dom.Text.Text:
    assert( numChildren == 0 )
    assert( not attrs )
    processContent( curNode.nodeValue, ordpath )
elif not attrs and numChildren == 1 and \
    curNode.childNodes[ childIndices[ 0 ] ].__class__ == xml.dom.Text.Text:
    # simple element node (e.g., <element>text</element>)
    processElement( curNode.nodeName, attrs, walker.nextNode().nodeValue,
                    ordpath, tagdict )
elif not attrs and numChildren == 0:
    # simple empty element node (e.g., <element></element>)
    processElement( curNode.nodeName, attrs, None, ordpath, tagdict )
else:
    # non-simple element node
    processElement( curNode.nodeName, attrs, None, ordpath, tagdict )
    if numChildren:
        if not attrs:
            ordpath.append( -1 )
        for i in range( numChildren ):
            if walker.nextNode():
                ordpath.incrementLastComponent( 2 )
                processNode( walker, ordpath, tagdict )
            else:
                raise ValueError("walker.nextNode() returned False when
                expecting child nodes."")
        ordpath.dropLastComponent()
elif attrs:
    ordpath.dropLastComponent()
curDepth -= 1

class filter(object):
    def getAcceptedChildrenIndices( node ):
        indices = []
        for i in range( len( node.childNodes )):
            if filter.acceptNode( node.childNodes[i] ) == NodeFilter.FILTER_ACCEPT:
                indices.append( i )
        return indices
getAcceptedChildrenIndices = staticmethod( getAcceptedChildrenIndices )
def acceptNode( node ):
    if checkProcessingInstruction: # global variable indicating presence of
    # xml.dom.ProcessingInstruction
if node.__class__ == \\n    xml.dom.ProcessingInstruction.ProcessingInstruction:
    return NodeFilter.FILTER_REJECT
if node.__class__ == xml.dom.Text.Text:
    if not node.nodeValue.strip():
        return NodeFilter.FILTER_REJECT
return NodeFilter.FILTER_ACCEPT

acceptNode = staticmethod( acceptNode )

if __name__ == '__main__':
    if len( sys.argv ) != 3:
        print "Usage: oshred.py filename tablename"
        sys.exit(1)
    try:
        f = file( sys.argv[1] )
    except Exception, err:
        print "Error: cannot open file: " + sys.argv[1]
        sys.exit(1)

    def doShred():
        global checkProcessingInstruction, checkComment, cursor, tablename
        conn = getConnection()

        # initialize variables
        reader = Sax2.Reader()
        doc = reader.fromStream( f )
        dirXmlDom = dir( xml.dom )
        checkProcessingInstruction = \
            dirXmlDom.__contains__('ProcessingInstruction')
        checkComment = dirXmlDom.__contains__( 'Comment' )
        cursor = conn.cursor()
        tablename = sys.argv[2]
        tagdict = tagDict()
        ordpath = ordPath( cursor )
        walker = doc.createTreeWalker( doc.documentElement, \
            NodeFilter.SHOW_ALL, filter, 0 )

        cursor.execute( "USE ordpath;" )
        cursor.execute( "DROP TABLE IF EXISTS " + tablename + ";" )
        cursor.execute( "DROP TABLE IF EXISTS " + tablename + "._tags" + ";" )
        cursor.execute( """CREATE TABLE %s( ordpath TEXT, tag INT, \
            nodeType INT, value TEXT, depth INT, \
            PRIMARY KEY( ordpath(255)), \
            INDEX( depth ), \
            INDEX( nodeType ));"" % tablename )
        cursor.execute( """CREATE TABLE %s_tags( tagName TEXT, tagNum INT, \
            PRIMARY KEY( tagNum ), \
            UNIQUE( tagName(255)))"" % tablename )

        ordpath.append( 1 )
        processNode( walker, ordpath, tagdict )
        tagdict.saveToDatabase( cursor, tablename + "._tags" )
        cursor.execute( "COMMIT;" )
        cursor.close()
        conn.close()
        doProfileAndPrintStats( doShred, None )
A.3 Abstract Base Traverser Class

# --------------------------------------------------------------
# abstractTraverser.py
# ---------------------
# The application programming interface to support queries
# against underlying shredded XML documents. Classes to
# support querying specific shredding methods should
# implement this interface.
# --------------------------------------------------------------

class abstractTraverser( object ):
    ELEMENT_NODE_TYPE = 1
    ATTRIBUTE_NODE_TYPE = 2
    COMMENT_NODE_TYPE = 3
    VALUE_NODE_TYPE = 4
    ATTR_VALUE_NODE_TYPE = 5  # used only to distinguish between content and
    # attribute values types for
    # getDescendantsByNameAndNodeType and
    # getNthGenerationDescendantsByNameAndNodeType

    def validateXPath( xpath ):
        if len(xpath) < 2:
            return False, 'Must be at least 2 characters long.'
        if xpath[0] != '/':
            return False, "Must begin with '/'"  
        return True, ''

    def getNodeAtXPath( self, xpath ):
        res, msg = self.validateXPath( xpath )
        if not res:
            raise ValueError( 'Error: Invalid XPath: %s: %s.' % ( xpath, msg ) )

        levels = xpath.split( '/' )[2:]
        curNode = self.getRootNode()

        for l in [ k for k in levels if k != '']:
            if curNode == None:
                break
            curNode = self.getFirstChildByName( curNode, l )
        return curNode

    # derived classes can supply a better version of this
def getFirstChildByName( self, node, name ):
        child = self.getFirstChild( node )
        while( child and self.getNodeName( child ) != name ):
            child = self.getNextSiblingByName( child, name )
        return child

    # derived classes can supply a better version of this
    # 1-based 
    def getNthChildByName( self, node, n, name ):
        count = 0
        child = self.getFirstChildByName( node, name )
while( child != None ):
count += 1
if count == n:
    break
child = self.getNextSiblingByName( child, name )
return child

# derived classes can supply a better version of this
def getLastChildByName( self, node, name ):
    child = self.getLastChild( node )
    while( child != None ):
        if self.getNodeName( child ) == name:
            break
        child = self.getPreviousSibling( child )
    return child

# derived classes can supply a better version of this
def getChildrenByName( self, node, name ):
    retVal = []
    child = self.getFirstChildByName( node, name )
    while( child != None ):
        retVal.append( child )
        child = self.getNextSiblingByName( child, name )
    return retVal

# derived classes may have ability to define a better version of this.
def getFirstChildWithAttributeValue( self, node, child, attribute, value ):
    curChild = self.getFirstChildByName( node, child )
    while( curChild and
        self.getNodeAttributeValue( curChild, attribute ) != value ):
        curChild = self.getNextSiblingByName( curChild, child )
    return curChild

# derived classes can define a better version of this.
def getNextSiblingByName( self, node, name ):
    curSibling = self.getNextSibling( node )
    while( curSibling and self.getNodeName( curSibling ) != name ):
        curSibling = self.getNextSibling( curSibling )
    return curSibling

def insertSimpleNode( self, leftSibling, newNode ): abstract
def getRootNode( self ): abstract
def getRandomNonrootNode( self ): abstract
def getType( self, node ): abstract
def getNode( self, node ): abstract
def getText( self, node ): abstract
def getAttribute( self, node, attribute ): abstract
def getParent( self, node ): abstract
def getNextSibling( self, node ): abstract
def getPreviousSibling( self, node ): abstract
def getFirstChild( self, node ): abstract
def getLastChild( self, node ): abstract
def getDescendants( self, node ): abstract
def getDescendantsContainingText( self, node, text ): abstract
def getDescendantsByNameAndNodeType( self, node, name, nodeType ): abstract
def getNthGenerationDescendantsByNameAndNodeType( self, node, n, name, nodeType ): abstract
abstract
def getAncestors( self, node ): abstract
def compareNodes( self, node1, node2 ): abstract
A.4 Method-Specific Derivations of AbstractTraverser

A.4.1 Interval Implementation

# -------------------------------------------------------------
# ilib.py
# -------
# Interval-specific implementation of abstractTraverser.
# -------------------------------------------------------------

import MySQLdb
from abstractTraverser import *

# ---------------------------------------------------------------
# Helper functions
# ---------------------------------------------------------------

def createRootOperator( cursor, tablename ):
    cursor.execute( 'DROP VIEW IF EXISTS %s_root;' % tablename )
    s = """CREATE VIEW %s_root AS
        SELECT u.s as s, u.l as l, u.r as r
        FROM %s u
        WHERE NOT EXISTS(
            SELECT *
            FROM %s v
            WHERE v.l < u.l AND u.r < v.r );""" % 
            (tablename, tablename, tablename)
    cursor.execute( s )

def createChildrenOperator( cursor, tablename ):
    cursor.execute( 'DROP VIEW IF EXISTS %s_children;' % tablename )
    s = """CREATE VIEW %s_children AS
        SELECT u.s as s, u.l as l, u.r as r
        FROM %s u
        WHERE EXISTS(
            SELECT *
            FROM %s v
            WHERE v.l < u.l AND u.r < v.r );""" % 
            (tablename, tablename, tablename)
    cursor.execute( s )

# -----------------------
# Tree-traversal
# -----------------------
class intervalTraverser( abstractTraverser ):
    S_INDEX = 0
    L_INDEX = 1
    R_INDEX = 2
    NODE_TYPE_INDEX = 3
    DEPTH_INDEX = 4

    def __init__( self, cursor, tablename ):
        self.cursor = cursor
        selftablename = tablename

    def compareNodes( self, node1, node2 ):
return node1[self.L_INDEX].__cmp__( node2[self.L_INDEX] )

def insertSimpleNode( self, leftSibling, node ):
    self.cursor.execute( "UPDATE %s SET l = l + 2, r = r + 2 WHERE l > %s
                    ORDER BY l DESC;" % 
        ( self.tablename, leftSibling[self.R_INDEX] )
    self.cursor.execute( 'INSERT INTO %s VALUES( %s, %s, %s, %s, %s);' % 
        ( self.tablename, node, leftSibling[self.R_INDEX] + 1,
         leftSibling[self.R_INDEX] + 2,
         self.ELEMENT_NODE_TYPE, leftSibling[self.DEPTH_INDEX] )
    parent = self.getParent( leftSibling )
    while( parent ):
        self.cursor.execute("UPDATE %s SET r = r + 2 WHERE l = %s;" % 
        ( self.tablename, parent[self.L_INDEX] )
        parent = self.getParent( parent )
    self.cursor.execute( "COMMIT;" )

def getRootNode( self ):
    self.cursor.execute( "SELECT * FROM %s WHERE l = 0;" % self.tablename )
    return self.cursor.fetchone()

def getRandomNonrootNode( self ):
    self.cursor.execute( """SELECT * FROM %s WHERE nodeType = %s AND l > 0
                           ORDER BY RAND() LIMIT 1;""" % 
        ( self.tablename, self.ELEMENT_NODE_TYPE )
    return self.cursor.fetchone()

def getNodeName( self, node ):
    return node[self.S_INDEX]

def getNodeText( self, node ):
    if node == None: return ["""]
    self.cursor.execute( """SELECT * FROM %s WHERE l > %s AND l < %s AND
                           depth = %s AND nodeType = %s;""" % 
        ( self.tablename, node[self.L_INDEX], node[self.R_INDEX],
         node[self.DEPTH_INDEX] + 1, self.VALUE_NODE_TYPE )
    res = self.cursor.fetchall()
    return [x[self.S_INDEX] for x in res]

def getNodeAttributeValue( self, node, attribute ):
    attrIndex = node[self.L_INDEX] + 1
    while 1:
        self.cursor.execute("SELECT * FROM %s WHERE l=%s AND nodeType=%s;" % 
        ( self.tablename, attrIndex, self.ATTRIBUTE_NODE_TYPE )
        res = self.cursor.fetchone()
        if not res: return None
        if res[self.S_INDEX] == attribute:
            self.cursor.execute( "SELECT * FROM %s WHERE l = %s;" % 
            ( self.tablename, res[self.L_INDEX] + 1 )
            res = self.cursor.fetchone()
            return res[self.S_INDEX]
        attrIndex += 4
    
def getNextSibling( self, node ):
    self.cursor.execute( "SELECT * FROM %s WHERE l=%s AND nodeType=%s;" % 
        ( self.tablename, node[self.R_INDEX] + 1, self.ELEMENT_NODE_TYPE )
    res = self.cursor.fetchone()
if res == None:
    # try skipping a VALUE node (text content for parent element) to
    # find next sibling
    self.cursor.execute( """SELECT t.* FROM(  
            SELECT * FROM %s WHERE l = %s AND  
            nodeType = %s ) t """" % 
        ( self.tablename, node[self.R_INDEX] + 3,  
        self.ELEMENT_NODE_TYPE ) + 
    """WHERE EXISTS(  
            SELECT * FROM %s  
            WHERE l = t.l - 2 AND nodeType = %s );""" % 
    ( self.tablename, self.VALUE_NODE_TYPE ))
    res = self.cursor.fetchone()
return res

def getNextSiblingByNameAlt( self, node, name ):
    self.cursor.execute( """SELECT * FROM %s  
        WHERE l > %s AND nodeType = %s AND depth = %s  
        AND s = '%s' ORDER BY l LIMIT 1;""" % 
    ( self.tablename, node[self.L_INDEX], self.ELEMENT_NODE_TYPE,  
    node[self.DEPTH_INDEX], name ))
    return self.cursor.fetchone()

def getPreviousSibling( self, node ):
    self.cursor.execute( """SELECT * FROM %s  
        WHERE r = %s AND nodeType = %s;""" % 
    ( self.tablename, node[self.L_INDEX] - 1, self.ELEMENT_NODE_TYPE ))
    res = self.cursor.fetchone()
    if res == None:
        # try skipping a VALUE node (text content for the parent element) to
        # find prev sibling
        self.cursor.execute( """SELECT t.* FROM(  
            SELECT * FROM %s  
            WHERE r=%s AND nodeType = %s ) t"""" % 
        ( self.tablename, node[self.L_INDEX] - 3,  
        self.ELEMENT_NODE_TYPE ) + 
    """ WHERE EXISTS(  
            SELECT * FROM %s WHERE l=t.l+2 AND nodeType = %s );""" % 
    ( self.tablename, self.VALUE_NODE_TYPE ))
    return res

def getFirstChild( self, node ):
    sql = """SELECT * FROM %s  
        WHERE l > %s AND l < %s AND nodeType = %s  
        ORDER BY l LIMIT 1;""" % 
    ( self.tablename, node[self.L_INDEX],  
    node[self.R_INDEX], self.ELEMENT_NODE_TYPE )
    self.cursor.execute( sql )
    return self.cursor.fetchone()

def getFirstChildByName( self, node, name ):
    self.cursor.execute( "SELECT * FROM %s " % self.tablename + 
    " WHERE l > %s AND l < %s AND nodeType = %s AND s = '%s'  
    AND depth = %s ORDER BY l LIMIT 1;""" % 
    ( node[self.L_INDEX], node[self.R_INDEX],  
    self.ELEMENT_NODE_TYPE, name, node[self.DEPTH_INDEX] + 1 ))
    return self.cursor.fetchone()
def getNthChildByName( self, node, n, name ):
    self.cursor.execute( "SELECT * FROM %sаблицу + \n    """WHERE l > %s AND l < %s AND nodeType = %s AND s = '%s' AND depth = %s ORDER BY l LIMIT 1 OFFSET %s;"" % 
    ( node[self.L_INDEX], node[self.R_INDEX], self.ELEMENT_NODE_TYPE, 
    name, n - 1, node[self.DEPTH_INDEX] + 1 ))
    return self.cursor.fetchone()

def getParent( self, node ):
    self.cursor.execute( "SELECT * FROM %s WHERE l = " % self.tablename + \n    """( SELECT MAX(l) FROM %s WHERE l <= %s AND depth = %s AND nodeType = %s );" % 
    ( self.tablename, node[self.L_INDEX], node[self.DEPTH_INDEX] - 1, 
    self.ELEMENT_NODE_TYPE))
    return self.cursor.fetchone()

def getDescendants( self, node ):
    self.cursor.execute( "SELECT * FROM %s WHERE l >= %s AND l <= %s;" % 
    ( self.tablename, node[self.L_INDEX], node[self.R_INDEX] ))
    return self.cursor.fetchall()

def getDescendantsContainingText( self, node, text ):
    self.cursor.execute( "SELECT * FROM %s WHERE l >= %s AND l <= %s AND s LIKE '%%%s%%';" % 
    ( self.tablename, node[self.L_INDEX], node[self.R_INDEX], text ))
    return self.cursor.fetchall()

def getDescendantsByNameAndNodeType( self, node, name, nodeType ):
    if nodeType == self.ATTR_VALUE_NODE_TYPE:
        nodeType = self.VALUE_NODE_TYPE
    self.cursor.execute( "SELECT * FROM %s WHERE l > %s AND l < %s AND s = '%s' AND nodeType = %s;" % 
    ( self.tablename, node[self.L_INDEX], node[self.R_INDEX], name, nodeType )
    return self.cursor.fetchall()

def getNthGenerationDescendantsByNameAndNodeType( self, node, n, name, nodeType ):
    if nodeType == self.ATTR_VALUE_NODE_TYPE:
        nodeType = self.VALUE_NODE_TYPE
    self.cursor.execute( "SELECT * FROM %s WHERE l > %s AND l < %s AND nodeType = %s AND s = '%s' AND depth = %s;" % 
    ( node[self.L_INDEX], node[self.R_INDEX], nodeType, name, 
    node[self.DEPTH_INDEX] + n ))
    return self.cursor.fetchall()

def getLastChild( self, node ):
    self.cursor.execute("SELECT * FROM %s WHERE r=%s AND nodeType=%s;" % 
    ( self.tablename, node[self.R_INDEX] - 1, self.ELEMENT_NODE_TYPE ))
    return self.cursor.fetchone()

def getLastChildByName( self, node, name ):
    self.cursor.execute("SELECT * FROM %s WHERE l=" % (self.tablename) + \n    """( SELECT MAX(l) FROM %s WHERE r < %s AND nodeType = %s AND s = '%s' );" % 
    ( self.tablename, node[self.R_INDEX] - 1, self.ELEMENT_NODE_TYPE, 
    name, node[self.DEPTH_INDEX] + 1 ))
    return self.cursor.fetchone()
(self.tablename, node[self.R_INDEX], self.ELEMENT_NODE_TYPE, name ))
return self.cursor.fetchone()

def getChildrenByName( self, node, name ):
    self.cursor.execute( "**SELECT * FROM %s WHERE l > %s AND l < %s
    AND nodeType = %s AND depth = %s AND s = '%s';" % 
        ( self.tablename, node[self.L_INDEX], node[self.R_INDEX],
          self.ELEMENT_NODE_TYPE, node[self.DEPTH_INDEX] + 1, name ))
    return self.cursor.fetchall()

def getAncestors( self, node ):
    self.cursor.execute( "SELECT * FROM %s WHERE l < %s AND r > %s;" % 
        ( self.tablename, node[self.L_INDEX], node[self.R_INDEX] ))
    return cursor.fetchall()
A.4.2 ORDPATH Implementation

# ------------------------------------------------------------
# olib.py
# -------
# ORDPATH-specific implementation of abstractTraverser.
# ------------------------------------------------------------

import MySQLdb
import sys
import copy
from abstractTraverser import *

sys.path.append( '../ordpath' )
from ordpath import *

# -----------------------
# Tree-traversal
# -----------------------

class ordpathTraverser( abstractTraverser ):
    rootHexstring = None
    HEXSTRING_PATH_INDEX = 0
    TAG_INDEX = 1
    NODE_TYPE_INDEX = 2
    VALUE_INDEX = 3
    DEPTH_INDEX = 4
    ORDPATH_INDEX = 5

    def __init__( self, cursor, tablename ):
        self.cursor = cursor
        self.tablename = tablename

        if self.rootHexstring == None:
            o = ordPath( self.cursor )
            o.append( 1 )
            self.rootHexstring = o.getHexstringRepr()

    def getOrCreateOrdpath( self, node ):
        if len(node) == 6:
            return node[self.ORDPATH_INDEX]
        else:
            o = ordPath( self.cursor )
            o.setFromHexstring( node[self.HEXSTRING_PATH_INDEX] )
            node = ( node[0], node[1], node[2], node[3], node[4], o )
            return o, node

    def compareNodes( self, node1, node2 ):
        if node1[self.HEXSTRING_PATH_INDEX] == node2[self.HEXSTRING_PATH_INDEX]:
            return 0
        elif node1[self.HEXSTRING_PATH_INDEX] < node2[self.HEXSTRING_PATH_INDEX]:
            return 1
        else:
            return -1
def insertSimpleNode( self, leftSibling, node ):
    nextSib = self.getNextSibling( leftSibling )
    leftSibOrdpath, leftSibling = self.getOrCreateOrdpath( leftSibling )
    if nextSib:
        nextSibOrdpath, nextSib = self.getOrCreateOrdpath( nextSib )
        if len( leftSibOrdpath.getListRepr() ) == \
            len( nextSibOrdpath.getListRepr() ):
            newNodeOrdpath = copy.copy( leftSibOrdpath )
            newNodeOrdpath.incrementLastComponent( 1 )
            newNodeOrdpath.append( 1 )
        else:
            newNodeOrdpath = copy.copy( nextSibOrdpath )
            newNodeOrdpath.incrementLastComponent( 2 )
    else:
        newNodeOrdpath = copy.copy( leftSibOrdpath )
        newNodeOrdpath.incrementLastComponent( -2 )
    tag = self.getTagFromIdentifier( node )
    if not tag:
        self.cursor.execute( "SELECT MAX(tagNum) + 1 FROM %s_tags;" % ( self.tablename ))
        tag = self.cursor.fetchone()[0]
        self.cursor.execute( "INSERT INTO %s_tags VALUES( '%s', %s );" % ( self.tablename, node, tag ))
        self.cursor.execute("INSERT INTO %s VALUES('%s',%s, %s, NULL, %s);" % ( self.tablename, newNodeOrdpath.getHexstringRepr(), tag,
        self.ELEMENT_NODE_TYPE, leftSibling[self.DEPTH_INDEX] ))
        self.cursor.execute( "COMMIT;" )

def getRootNode( self ):
    self.cursor.execute( "SELECT * FROM %s WHERE ordpath = '%s'" % ( self.tablename, self.rootHexstring ))
    return self.cursor.fetchone()

def getRandomNonrootNode( self ):
    self.cursor.execute( """SELECT * FROM %s WHERE nodeType = %s AND ordpath != '%s' ORDER BY RAND() LIMIT 1;" % ( self.tablename, self.ELEMENT_NODE_TYPE, self.rootHexstring )
    return self.cursor.fetchone()

def getNodeText( self, node ):
    if node == None: return ['''']
    if node[self.VALUE_INDEX] != 'NULL':
        return [node[self.VALUE_INDEX]]
    else:
        upperBound, node = self.getOrCreateOrdpath( node )
        upperBound.incrementLastComponent( 1 )
        upperBound = upperBound.getHexstringRepr()
        self.cursor.execute( "SELECT * FROM %s WHERE nodeType = %s AND " % ( self.tablename, self.VALUE_NODE_TYPE ) + \" ordpath > '%s' AND ordpath < '%s' AND depth = %s;" % ( node[self.HEXSTRING_PATH_INDEX], upperBound, node[self.DEPTH_INDEX] + 1 ))
        res = self.cursor.fetchall()
        # reset original node ordPath object
        node[self.ORDEP_PATH_INDEX].incrementLastComponent( -1 )
        return [x[self.VALUE_INDEX] for x in res]
def getNextSibling( self, node):
    o, node = self.getOrCreateOrdpath( node )
    upperBound = copy.copy( o )
    upperBound.dropLastComponentIncludeEvens()
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    self.cursor.execute( "**SELECT * 
    FROM %s
    WHERE ordpath > '%s' AND ordpath < '%s'
    AND depth = %s
    ORDER BY ordpath LIMIT 1;" % 
    ( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
      upperBound, node[self.DEPTH_INDEX] ))
    res = self.cursor.fetchone()
    return res

def getNextSiblingByName( self, node, name):
    o, node = self.getOrCreateOrdpath( node )
    upperBound = copy.copy( o )
    upperBound.dropLastComponentIncludeEvens()
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    tag = self.getTagFromIdentifier( name )
    if not tag: return None
    self.cursor.execute( "**SELECT * 
    FROM %s
    WHERE ordpath > '%s' AND ordpath < '%s'
    AND depth = %s
    AND tag = %s ORDER BY ordpath LIMIT 1;" % 
    ( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
      upperBound, node[self.DEPTH_INDEX], tag ))
    res = self.cursor.fetchone()
    return res

def getPreviousSibling( self, node):
    lowerBound = copy.copy( o )
    lowerBound.dropLastComponentIncludeEvens()
    lowerBound = lowerBound.getHexstringRepr()
    self.cursor.execute( "**SELECT * 
    FROM %s
    WHERE ordpath = ( 
        SELECT MAX( ordpath ) 
    FROM %s
    WHERE ordpath > '%s' AND ordpath < '%s'
    AND depth = %s );" % 
    ( self.tablename, self.tablename, lowerBound, \
      node[self.HEXSTRING_PATH_INDEX], node[self.DEPTH_INDEX] ))
    res = self.cursor.fetchone()
    return res

def getFirstChild( self, node):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    self.cursor.execute( "**SELECT * 
    FROM %s
    WHERE ordpath > '%s' AND ordpath < '%s'
    AND depth = %s
    ORDER BY ordpath LIMIT 1;" % 
    ( self.tablename, upperBound, node[self.DEPTH_INDEX] ))
    res = self.cursor.fetchone()
    return res
FROM %s
WHERE ordpath > '%s' AND ordpath < '%s'
AND depth = %s
ORDER BY ordpath LIMIT 1;""
( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
upperBound, node[self.DEPTH_INDEX] + 1 ))
res = self.cursor.fetchone()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
return res
def getFirstChildByName( self, node, name ):
upperBound, node = self.getOrCreateOrdpath( node )
upperBound.incrementLastComponent( 1 )
upperBound = upperBound.getHexstringRepr()
tag = self.getTagFromIdentifier( name )
if not tag: return None
self.cursor.execute( """SELECT *
FROM %s
WHERE ordpath > '%s' AND ordpath < '%s'
AND depth = %s
AND tag = %s
ORDER BY ordpath LIMIT 1;""
( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
upperBound, node[self.DEPTH_INDEX] + 1, tag ))
res = self.cursor.fetchone()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
return res
def getNthChildByName( self, node, n, name ):
upperBound, node = self.getOrCreateOrdpath( node )
upperBound.incrementLastComponent( 1 )
upperBound = upperBound.getHexstringRepr()
tag = self.getTagFromIdentifier( name )
if not tag: return None
self.cursor.execute( """SELECT *
FROM %s
WHERE ordpath > '%s' AND ordpath < '%s'
AND depth = %s
AND tag = %s
ORDER BY ordpath LIMIT 1 OFFSET %s;""
( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
upperBound, node[self.DEPTH_INDEX] + 1, tag, n - 1 ))
res = self.cursor.fetchone()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
return res
def getChildrenByName( self, node, name ):
upperBound, node = self.getOrCreateOrdpath( node )
upperBound.incrementLastComponent( 1 )
upperBound = upperBound.getHexstringRepr()
tag = self.getTagFromIdentifier( name )
if not tag: return None
self.cursor.execute( """SELECT *
FROM %s
WHERE ordpath > '%s' AND ordpath < '%s'
AND depth = %s
AND tag = %s
ORDER BY ordpath LIMIT 1;""
( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
upperBound, node[self.DEPTH_INDEX] + 1, tag, n - 1 ))
res = self.cursor.fetchone()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
return res
WHERE ordpath > '%s' AND ordpath < '%s'
AND depth = %s
AND tag = %s;

( self.tablename, node[self.HEXSTRING_PATH_INDEX], \
  upperBound, node[self.DEPTH_INDEX] + 1, tag ))
res = self.cursor.fetchall()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent(-1)
return res
def getLastChild( self, node ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    self.cursor.execute( """SELECT *
        FROM %s
        WHERE ordpath = ("""
        SELECT MAX( ordpath )
        FROM %s
        WHERE ordpath > '%s' AND ordpath < '%s'
        AND depth = %s )"""
        ( self.tablename, self.tablename, node[self.HEXSTRING_PATH_INDEX], \
          upperBound, node[self.DEPTH_INDEX] + 1 ))
    res = self.cursor.fetchone()
    # reset original node ordpath object
    node[-1].incrementLastComponent(-1)
    return res
def getLastChildByName( self, node, name ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    tag = self.getTagFromIdentifier( name )
    if not tag: return None
    self.cursor.execute( """SELECT *
        FROM %s
        WHERE ordpath = ("""
        SELECT MAX( ordpath )
        FROM %s
        WHERE ordpath > '%s' AND ordpath < '%s'
        AND depth = %s AND tag = %s )"""
        ( self.tablename, self.tablename, node[self.HEXSTRING_PATH_INDEX], \
          upperBound, node[self.DEPTH_INDEX] + 1, tag ))
    res = self.cursor.fetchone()
    # reset original node ordpath object
    node[self.ORDPATH_INDEX].incrementLastComponent(-1)
    return res
def getIdentifierFromTag( self, tag ):
    self.cursor.execute( """SELECT * FROM %s_tags WHERE tagNum = %s;"""
    ( self.tablename, tag ))
    res = self.cursor.fetchone()
    if res:
        res = res[0]
    return res
def getTagFromIdentifier( self, id ):
    self.cursor.execute ( "SELECT * FROM %s_tags WHERE tagName = '%s';" %
        ( self.tablename, id ))
    res = self.cursor.fetchone()
    if res:
        res = res[1]
    return res

def getNodeName( self, node ):
    return self.getIdentifierFromTag( node[self.TAG_INDEX] )

def getParent( self, node ):
    self.cursor.execute("SELECT * FROM %s WHERE ordpath = ( " %
        ( self.tablename ) + "SELECT MAX(ordpath) FROM %s WHERE ordpath<'%s' AND depth=%s ;" %
        ( self.tablename, node[self.HEXSTRING_PATH_INDEX], node[self.DEPTH_INDEX] - 1 ))
    return self.cursor.fetchone()

def getNodeAttributeValue( self, node, attribute ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    attributeTag = self.getTagFromIdentifier( attribute )
    self.cursor.execute( """SELECT * FROM %s WHERE tag = %s
        AND nodeType = %s AND """ %
        ( self.tablename, attributeTag, self.ATTRIBUTE_NODE_TYPE ) + "
        ordpath > '%s' AND ordpath < '%s' AND depth = %s;" %
        ( node[self.HEXSTRING_PATH_INDEX], upperBound, 
          node[self.DEPTH_INDEX]))
    res = self.cursor.fetchone()
    if res:
        res = res[self.VALUE_INDEX]
    # reset original node ordpath object
    node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
    return res

def getDescendants( self, node ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    self.cursor.execute("""SELECT * FROM %s WHERE ordpath > '%s'
        AND ordpath < '%s';"""
        ( self.tablename, node[self.HEXSTRING_PATH_INDEX], upperBound ))
    res = self.cursor.fetchall()
    # reset original node ordPath object
    node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
    return res

def getDescendantsContainingText( self, node, text ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    tag = self.getTagFromIdentifier( text )
if not tag:
    tag = "value LIKE '%%%s%%'" % text
else:
    tag="tag in (SELECT * from %s_tags WHERE tagName LIKE '%%%s%%')" % \
    ( self.tablename, tag )

sql = """SELECT * FROM %s WHERE ordpath > '%s' AND ordpath < '%s'
AND %s;""" % \
    (self.tablename, node[self.HEXSTRING_PATH_INDEX], upperBound, tag )
self.cursor.execute( sql )
res = self.cursor.fetchall()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
return res

def getDescendantsByNameAndNodeType( self, node, name, nodeType ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()
    tag = self.getTagFromIdentifier( name )
    if not tag:
        if nodeType != self.VALUE_NODE_TYPE:
            return []
        else:
            tag = "tag = %s" % tag
    else:
        tag = 'tag = %s' % tag

    self.cursor.execute( """SELECT * FROM %s WHERE ordpath > '%s'
AND ordpath < '%s'""" % \
    ( self.tablename, node[self.HEXSTRING_PATH_INDEX], upperBound ) + \
    " AND nodeType = %s AND %s;" % \
    ( nodeType, tag ) )

    res = self.cursor.fetchall()
    # reset original node ordPath object
    node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
    return res

def getNthGenerationDescendantsByNameAndNodeType( self, node, n, name, \
    nodeType ):
    upperBound, node = self.getOrCreateOrdpath( node )
    upperBound.incrementLastComponent( 1 )
    upperBound = upperBound.getHexstringRepr()

    if nodeType == self.ATTR_VALUE_NODE_TYPE:
        nodeType = self.ATTRIBUTE_NODE_TYPE
        tag = "value = '%s' AND*" % name
    else:
        tag = self.getTagFromIdentifier( name )
        if not tag:
            if nodeType != self.VALUE_NODE_TYPE:
                return None
            else:
                tag = "tag = %s AND' % tag

    self.cursor.execute( """SELECT * FROM %s WHERE ordpath > '%s'
AND ordpath < '%s'""" % \
    ( self.tablename, node[self.HEXSTRING_PATH_INDEX], upperBound ) ) + \
    " AND nodeType = %s AND %s;" % \
    ( nodeType, tag ) )

    res = self.cursor.fetchall()
    # reset original node ordPath object
    node[self.ORDPATH_INDEX].incrementLastComponent( -1 )
    return res

79
self.cursor.execute("\n**SELECT * FROM %s WHERE ordpath > '%s'
    AND ordpath < '%s'**
    %
    ( selftablename, node[self.HEXSTRING_PATH_INDEX], upperBound ) + \n    " AND nodeType = %s AND %s depth = %s;" % \n    ( nodeType, tag, node[self.DEPTH_INDEX] + n ))

res = self.cursor.fetchall()
# reset original node ordPath object
node[self.ORDPATH_INDEX].incrementLastComponent(-1)

return res
import sys, os

sys.path.append('../shared')
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery('q1')
    def q1():
        parentNode = traverser.getNodeAtXPath('/site/people/)
        if parentNode:
            node = traverser.getFirstChildWithAttributeValue(parentNode, 
              'person', 'id', 'person0')

            if node:
                name = traverser.getFirstChildByName(node, 'name')
                print traverser.getNodeText(name)[0]

    doProfileAndPrintStats(q1, None)
    cursor.close()
    conn.close()
import sys, os

sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q2' )

    def processAuction( auction ):
        secBidder = traverser.getNthChildByName( auction, 2, 'bidder' )
        if not secBidder: return
        increase = traverser.getFirstChildByName( secBidder, 'increase' )
        if not increase: return
        print '<increase> %s </increase>' % 
            traverser.getNodeText( increase )[0]

    def q2():
        parentNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        if not parentNode: return
        auctions = traverser.getChildrenByName( parentNode, 'open_auction' )
        for auction in auctions:
            processAuction( auction )

doProfileAndPrintStats( q2, None )
cursor.close()
conn.close()
# Query 3

```python
import sys, os

sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q3' )

    def processAuction( auction ):
        firstBidder = traverser.getFirstChildByName( auction, 'bidder' )
        if not firstBidder: return
        firstIncrease = traverser.getFirstChildByName( firstBidder, 'increase' )
        if not firstIncrease: return
        lastBidder = traverser.getLastChildByName( auction, 'bidder' )
        if not lastBidder: return
        lastIncrease = traverser.getFirstChildByName( lastBidder, 'increase' )
        if not lastIncrease: return

        firstIncrease = traverser.getNodeText( firstIncrease )[0]
        lastIncrease = traverser.getNodeText( lastIncrease )[0]

        if eval( firstIncrease ) * 2 <= eval( lastIncrease ) :
            print '<increase first=%s
                  last=%s />' %
                  ( firstIncrease, lastIncrease )

    def q3():
        rootNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        if not rootNode: return
        auctions = traverser.getChildrenByName( rootNode, 'open_auction' )
        for auction in auctions:
            processAuction( auction )

    doProfileAndPrintStats( q3, None )
cursor.close()
conn.close()
```

---
# Query 4

import sys, os
from util import *
if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q4' )
    def q4():
        rootNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        auctions = traverser.getChildrenByName( rootNode, 'open_auction' )
        for auction in auctions:
            firstPersonRefs = \
                traverser.getNthGenerationDescendantsByNameAndNodeType( \
                    auction, 2, 'person120', traverser.ATTR_VALUE_NODE_TYPE )
            if firstPersonRefs:
                secondPersonRefs = \
                    traverser.getNthGenerationDescendantsByNameAndNodeType( \
                        auction, 2, 'person230', traverser.ATTR_VALUE_NODE_TYPE)
                if secondPersonRefs:
                    fprcob = None
                for fpr in firstPersonRefs:
                    parent = traverser.getParent( fpr )
                    if traverser.getNodeName( parent ) == 'bidder':
                        fprcob = fpr
                        break
                    if fprcob:
                        for spr in secondPersonRefs:
                            parent = traverser.getParent( spr )
                            if traverser.getNodeName( parent ) == 'bidder':
                                if traverser.compareNodes( fprcob, spr ) < 0:
                                    reserve = traverser.getFirstChildByName(auction, \
                                        'reserve' )
                                    print '<history> %s </history>' % \
                                        traverser.getNodeText( reserve )[0]
                                    break
    doProfileAndPrintStats( q4, None )
    cursor.close()
    conn.close()
import sys, os
sys.path.append('..//shared')
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery('q5')

    def q5():
        rootNode = traverser.getNodeAtXPath('/site/closed_auctions/')
        prices = traverser.getNthGenerationDescendantsByNameAndNodeType(rootNode, 2, 'price', traverser.ELEMENT_NODE_TYPE)

        for price in prices:
            priceText = traverser.getNodeText(price)[0]
            if eval(priceText) > 40:
                print priceText

    doProfileAndPrintStats(q5, None)
cursor.close()
conn.close()
import sys, os
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q6' )

    def q6():
        rootNode = traverser.getNodeAtXPath( '/site/regions/' )
        items = traverser.getDescendantsByNameAndNodeType( rootNode, 'item', traverser.ELEMENT_NODE_TYPE )

        print len(items)

    doProfileAndPrintStats( q6, None )
cursor.close()
conn.close()
# Query 7

```python
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q7' )

    def q7():
        rootNode = traverser.getNodeAtXPath( '/site/' )
        descriptions = traverser.getDescendantsByNameAndNodeType( rootNode, 'description', traverser.ELEMENT_NODE_TYPE )
        annotations = traverser.getDescendantsByNameAndNodeType( rootNode, 'annotation', traverser.ELEMENT_NODE_TYPE )
        emails = traverser.getDescendantsByNameAndNodeType(rootNode, 'email', traverser.ELEMENT_NODE_TYPE )
        print len( descriptions ) + len( annotations ) + len( emails )

doProfileAndPrintStats( q7, None )
cursor.close()
conn.close()
```
# Query 8

FOR $p$ IN document("auction.xml")/site/people/person
LET $a$ := FOR $t$ IN
document("auction.xml")/site/closed_auctions/closed_auction
    WHERE $t/buyer/@person = $p/@id
RETURN $t$
RETURN <item person=$p/name/text()> COUNT ($a) </item>

import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q8' )

    def q8():
        buyerIDs = []
        rootNode = traverser.getNodeAtXPath( '/site/people/' )
        people = traverser.getChildrenByName( rootNode, 'person' )
        rootNode = traverser.getNodeAtXPath( '/site/closed_auctions/' )
        auctions = traverser.getChildrenByName( rootNode, 'closed_auction' )
        for auction in auctions:
            buyer = traverser.getFirstChildByName( auction, 'buyer' )
            buyerIDs.append( traverser.getNodeAttributeValue( buyer, 'person' ))

        for person in people:
            matchedAuctions = []
            personName = traverser.getFirstChildByName( person, 'name' )
            personID = traverser.getNodeAttributeValue( person, 'id' )
            print '  <item person=%s> %s </item>' % 
                ( traverser.getNodeText( personName )[0], 
                    buyerIDs.count( personID ))

    doProfileAndPrintStats( q8, None )
cursor.close()
conn.close()
# Query 9

```python
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q9' )

    def q9():
        peopleRoot = traverser.getNodeAtXPath( './site/people/' )
        people = traverser.getChildrenByName( peopleRoot, 'person' )
        peopleIDs = []
        for person in people:
            peopleIDs.append( traverser.getNodeAttributeValue( person, 'id' ) )

        europeRoot = traverser.getNodeAtXPath( './site/regions/europe/' )
        europianItems = traverser.getChildrenByName( europeRoot, 'item' )
        europianItemIDs = []
        for item in europianItems:
            europianItemIDs.append( traverser.getNodeAttributeValue( item, 'id' ) )

        rootNode = traverser.getNodeAtXPath( './site/closed_auctions/' )
        itemrefs = traverser.getNthGenerationDescendantsByNameAndNodeType( \
            rootNode, 2, 'itemref', traverser.ELEMENT_NODE_TYPE )

        buyerIDs = []
        for buyer in buyers:
            buyerIDs.append( traverser.getNodeAttributeValue( buyer, 'person' ) )
        for i in range( len( peopleIDs ) - 1, -1, -1 ):
            if not buyerIDs.count( peopleIDs[i] ) > 0:
                peopleIDs.__delitem__( i )

        matches = {}
        itemIDsToNames = {}
        for itemref in itemrefs:
            itemID = traverser.getNodeAttributeValue( itemref, 'item' )
            if europianItemIDs.count( itemID ) > 0:
                auction = traverser.getParent( itemref )
                buyer = traverser.getFirstChildByName( auction, 'buyer' )
                buyerID = traverser.getNodeAttributeValue( buyer, 'person' )
```
if itemIDsToNames.keys().count(itemID) > 0:
    itemName = itemIDsToNames(itemID)
else:
    item = traverser.getFirstChildWithAttributeValue(europeRoot, 'item', 'id', itemID)
    itemName = traverser.getFirstChildByName(item, 'name')
    itemIDsToNames[itemID] = itemName
if matches.keys().count(buyerID) > 0:
    matches[buyerID].append(itemName)
else:
    matches[buyerID] = [itemName]

for id in peopleIDs:
    if matches.keys().count(id) > 0:
        person = traverser.getFirstChildWithAttributeValue(peopleRoot, 'person', 'id', id)
        personName = traverser.getFirstChildByName(person, 'name')
        personName = traverser.getNodeText(personName)[0]
        print '<person name=%s> ' % personName
        for item in matches[id]:
            print '<item> %s </item>' % item
        print '</person>'

doProfileAndPrintStats(q9, None)
cursor.close()
conn.close()
# Query 10

```python
if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q10' )

def q10():
    def getPersonsInterestCategories( person ):
        categories = []
        profile = traverser.getFirstChildByName( person, 'profile' )
        if profile != None:
            interests = traverser.getChildrenByName( profile, 'interest' )
            for interest in interests:
                category = traverser.getNodeAttributeValue(interest,'category')
                if not categories.__contains__( category ):
                    categories.append( category )
        return categories

    interestToPeopleMap = {}

    rootNode = traverser.getNodeAtXPath( '/site/people/' )
    people = traverser.getChildrenByName( rootNode, 'person' )
    for person in people:
        interests = getPersonsInterestCategories( person )
        for interest in interests:
            if not interestToPeopleMap.__contains__( interest ):
                interestToPeopleMap[ interest ] = []
            interestToPeopleMap[ interest ].append( person )
```

personsCategories = getPersonsInterestCategories(person)
for category in personsCategories:
    if not interestToPeopleMap.keys().__contains__(category):
        interestToPeopleMap[category] = [person]
    else:
        interestToPeopleMap[category].append(person)
for category in interestToPeopleMap.keys():
    print "<categorie>
    <id> %s </id>,
    for person in interestToPeopleMap[category]:
        stringArgs = (traverser.getNodeText(traverser.getFirstChildByName(person, 'gender')))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'age'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'education'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'income'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'name'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'street'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'city'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'country'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'email'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'homepage'))[0],
        traverser.getNodeText(traverser.getFirstChildByName(person, 'creditcard'))[0])
    print ""
    print  
    print "</categorie>
doProfileAndPrintStats(q10, None)
cursor.close()
conn.close()
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q11' )
    def q11():
        rootNode = traverser.getNodeAtXPath( '/site/people/' )
        people = traverser.getChildrenByName( rootNode, 'person' )
        rootNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        auctions = traverser.getChildrenByName( rootNode, 'open_auction' )
        initials = []
        for auction in auctions:
            initial = traverser.getFirstChildByName( auction, 'initial' )
            if initial != None:
                initial = eval( traverser.getNodeText( initial )[0] ) * 5000
                initials.append( initial )

        for person in people:
            matchedInitials = []
            name = traverser.getFirstChildByName( person, 'name' )
            profile = traverser.getFirstChildByName( person, 'profile' )
            if profile:
                income = traverser.getNodeAttributeValue( profile, 'income' )
                if income:
                    for initial in initials:
                        if eval( income ) > initial:
                            matchedInitials.append( initial )
                    print '<items name=%s> %s </items>' % ( traverser.getNodeText( name )[0], len( matchedInitials ))

doProfileAndPrintStats( q11, None )
cursor.close()
conn.close()
# Query 12

```
# FOR $p$ IN document("auction.xml")/site/people/person
# LET $l$ := FOR $i$ IN
document("auction.xml")/site/open_auctions/open_auction/initial
WHERE $p$/profile/@income > (5000 * $i$/text())
RETURN $i$
# WHERE $p$/profile/@income > 50000
# RETURN <items person=$p$/name/income/text()> COUNT ($l$) </person>
```

```python
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q12' )
    
    def q12():
        rootNode = traverser.getNodeAtXPath( '/site/people/' )
        people = traverser.getChildrenByName( rootNode, 'person' )
        rootNode = traverser.getNodeAtXPath( '/site/open_auctions/' )
        auctions = traverser.getChildrenByName( rootNode, 'open_auction' )
        initials = []
        for auction in auctions:
            initial = traverser.getFirstChildByName( auction, 'initial' )
            if initial != None:
                initial = eval( traverser.getNodeText( initial )[0] ) * 5000
                initials.append( initial )

        for person in people:
            matchedInitials = []
            name = traverser.getFirstChildByName( person, 'name' )
            profile = traverser.getFirstChildByName( person, 'profile' )
            if profile:
                income = eval( traverser.getNodeAttributeValue( profile, 'income' ))
                if income > 50000:
                    for initial in initials:
                        if income > initial:
                            matchedInitials.append( initial )
                    print '<items name=%s> %s </items>' % 
                        ( traverser.getNodeText( name )[0], \
                        len( matchedInitials ))

    doProfileAndPrintStats( q12, None )
    cursor.close()
    conn.close()
```
import sys, os

sys.path.append('..../shared')
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery('q13')

    def q13():
        rootNode = traverser.getNodeAtXPath('/site/regions/australia/');
        items = traverser.getChildrenByName( rootNode, 'item' )
        for item in items:
            name = traverser.getFirstChildByName( item, 'name' )
            desc = traverser.getFirstChildByName( item, 'description' )
            text = traverser.getFirstChild( desc )
            text = traverser.getNodeText( text )
            print '…<item name=%s> %s </item>' % ( traverser.getNodeText( name )[0], str.join( '
', text ))

    doProfileAndPrintStats( q13, None )
    cursor.close()
    conn.close()
# Query 14

```python
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q14' )

    def q14():
        count = 0
        rootNode = traverser.getNodeAtXPath( '/site/' )
        items = traverser.getDescendantsByNameAndNodeType( rootNode, 'item', traverser.ELEMENT_NODE_TYPE )
        for item in items:
            desc = traverser.getFirstChildByName( item, 'description' )
            descDescendants = traverser.getDescendantsContainingText( desc, 'gold' )
            if descDescendants:
                count += 1
                name = traverser.getFirstChildByName( item, 'name' )
                print traverser.getNodeText( name )[0]

doProfileAndPrintStats( q14, None )
cursor.close()
conn.close()
```
import sys, os

sys.path.append( './shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q15' )

    def q15():
        rootNode = traverser.getNodeAtXPath( '/site/closed_auctions/' )
auctions = traverser.getChildrenByName( rootNode, 'closed_auction' )
        for auction in auctions:
            annot = traverser.getFirstChildByName( auction, 'annotation' )
                if annot:
                    desc = traverser.getFirstChildByName( annot, 'description' )
                        if desc:
                            parlist = traverser.getFirstChildByName( desc, 'parlist' )
                                if parlist:
                                    listitems = traverser.getChildrenByName( parlist, 'listitem' )
                                        for listitem in listitems:
                                            parlist = traverser.getFirstChildByName( listitem, 'parlist' )
                                                if parlist:
                                                    innerListItems = traverser.getChildrenByName( parlist, 'listitem' )
                                                        for innerListItem in innerListItems:
                                                            text = traverser.getFirstChildByName( innerListItem, 'text' )
                                                                if text:
                                                                    emphs = traverser.getChildrenByName( text, 'emph' )
                                                                        for emph in emphs:
                                                                            keywords = traverser.getChildrenByName( emph, 'keyword' )
                                                                                for keyword in keywords:
                                                                                    print '<text> %s </text>' % str.join( '
\n', traverser.getNodeText( keyword ) )

doProfileAndPrintStats( q15, None )
cursor.close()
conn.close()
import sys, os

sys.path.append( './shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q16' )

    def processAuction( auction ):
        annot = traverser.getFirstChildByName( auction, 'annotation' )
        if annot:
            desc = traverser.getFirstChildByName( annot, 'description' )
            if desc:
                parlist = traverser.getFirstChildByName( desc, 'parlist' )
                if parlist:
                    listitems = traverser.getChildrenByName( parlist, 'listitem' )
                    for listitem in listitems:
                        parlist = traverser.getFirstChildByName( listitem, 'parlist' )
                        if parlist:
                            innerListItems = traverser.getChildrenByName( parlist, 'listitem' )
                            for innerListItem in innerListItems:
                                text = traverser.getFirstChildByName( innerListItem, 'text' )
                                if text:
                                    emphs = traverser.getChildrenByName( text, 'emph' )
                                    for emph in emphs:
                                        keywords = traverser.getChildrenByName( emph, 'keyword' )
                                        for keyword in keywords:
                                            keywordTxt = traverser.getNodeText( keyword )
                                            if keywordTxt:
                                                seller = traverser.getFirstChildByName( auction, 'seller' )
                                                if seller:
                                                    print '<person id=%s />' % 
                                                        traverser.getNodeAttributeValue( 
                                                            seller, 'person' )
                                                return

    def q16():
        rootNode = traverser.getNodeAtXPath( '/site/closed_auctions/' )
        auctions = traverser.getChildrenByName( rootNode, 'closed_auction' )
        for auction in auctions:
            processAuction( auction )
doProfileAndPrintStats( q16, None )
cursor.close()
conn.close()
# Query 17

```python
import sys, os

sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q17' )

    def q17():
        rootNode = traverser.getNodeAtXPath( '/site/people/' )
        persons = traverser.getChildrenByName( rootNode, 'person' )
        for person in persons:
            homepage = traverser.getFirstChildByName( person, 'homepage' )
            if not homepage:
                name = traverser.getFirstChildByName( person, 'name' )
                print '<person name=%s/>' % traverser.getNodeText( name )[0]

    doProfileAndPrintStats( q17, None )
    cursor.close()
    conn.close()
```
import sys, os

sys.path.append('..\shared')
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery('q18')

    def q18():
        rootNode = traverser.getNodeAtXPath('/site/open_auctions/')
        auctions = traverser.getChildrenByName(rootNode, 'open_auction')

        for auction in auctions:
            reserve = traverser.getFirstChildByName(auction, 'reserve')
            if reserve:
                print eval(traverser.getNodeText(reserve)[0]) * 2.20371

    doProfileAndPrintStats(q18, None)
    cursor.close()
    conn.close()
import sys, os
sys.path.append( '..\shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q19' )
    
    def q19():
        rootNode = traverser.getNodeAtXPath( '/site/regions/' )
        items = traverser.getDescendantsByNameAndNodeType( rootNode, 
            'item', traverser.ELEMENT_NODE_TYPE )

        results = {}

        for item in items:
            name = traverser.getFirstChildByName( item, 'name' )
            name = traverser.getNodeText( name )[0]
            location = traverser.getFirstChildByName( item, 'location' )
            location = traverser.getNodeText( location )[0]
            results[ name ] = location

        keys = results.keys()
        keys.sort()

        for k in keys:
            print '<item name=%s> %s </item>' % ( k, results[k] )

        doProfileAndPrintStats( q19, None )
    
cursor.close()
    conn.close()
# Query 20

```xml
<result>
  <preferred>
    COUNT(document("auction.xml")/site/people/person/profile[@income >= 100000])
  </preferred>,
  <standard>
    COUNT(document("auction.xml")/site/people/person/profile[@income < 100000
    and @income >= 30000])
  </standard>,
  <challenge>
    COUNT(document("auction.xml")/site/people/person/profile[@income < 30000])
  </challenge>,
  <na>
    COUNT (FOR $p in document("auction.xml")/site/people/person
    WHERE EMPTY($p/@income)
    RETURN $p)
  </na>
</result>
```

```python
import sys, os
sys.path.append( '../shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'q20' )
    def q20():
        rootNode = traverser.getNodeAtXPath( '/site/people/' )
        people = traverser.getChildrenByName( rootNode, 'person' )
        preferredCount = 0
        standardCount = 0
        challengeCount = 0
        naCount = 0
        for person in people:
            profile = traverser.getFirstChildByName( person, 'profile' )
            if profile:
                income = traverser.getNodeAttributeValue( profile, 'income' )
                if income:
                    income = eval( income )
                    if income >= 100000:
                        preferredCount += 1
                    elif income >= 30000:
                        standardCount += 1
                    elif income:
                        challengeCount += 1
                    else:
                        print 'Something went wrong.'
                else:
                    naCount += 1
```
else:
    naCount += 1

print ""
<result>
<preferred>
  %s
</preferred>,
<standard>
  %s
</standard>,
<challenge>
  %s
</challenge>,
<na>
  %s
</na>
</result>"" % (preferredCount, standardCount, challengeCount, naCount )

doProfileAndPrintStats( q20, None )
cursor.close()
conn.close()
A.6 Top-Level Insert Implementation

```python
import sys, os

sys.path.append( './shared' )
from util import *

if __name__ == '__main__':
    traverser, conn, cursor = setupForQuery( 'insert', 4, 
                                              'numberOfRepetitions' )
    numInserts = eval(sys.argv[3])

    def insert():
        for i in range( numInserts ):
            randNode = traverser.getRandomNonrootNode()
            traverser.insertSimpleNode( randNode, 'newElement' )

    doProfileAndPrintStats( insert, None )
```
VITA

JONATHAN L. LEONARD

Education: Public Schools, Blountville, TN.
East Tennessee State University, Johnson City, TN.