A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML

Ning Zhang
University of Waterloo
School of Computer Science
nzhang@uwaterloo.ca

Varun Kacholia*
Indian Institute of Technology, Bombay
Department of Computer Science
kacholia@iitb.ac.in

M. Tamer Özsu
University of Waterloo
School of Computer Science
tozsu@uwaterloo.ca

Abstract

Path expressions are ubiquitous in XML processing languages. Existing approaches evaluate a path expression by selecting nodes that satisfies the tag-name and value constraints and then joining them according to the structural constraints. In this paper, we propose a novel approach, next-of-kin (NoK) pattern matching, to speed up the node-selection step, and to reduce the join size significantly in the second step. To efficiently perform NoK pattern matching, we also propose a succinct XML physical storage scheme that is adaptive to updates and streaming XML as well. Our performance results demonstrate that the proposed storage scheme and path evaluation algorithm is highly efficient and outperforms the other tested systems in most cases.

1. Introduction

The increasingly wider use of XML has heightened the need to store large volumes of data encoded in XML, and to query XML data more efficiently. Ubiquitously presenting itself as part of XML processing languages (XQuery [4], etc.), path expressions are arguably the most natural way to query tree-structured data such as XML trees.

The problem of evaluating path expressions against XML trees can be modeled as the *tree pattern matching* (TPM) problem [15]: a path expression can be represented as a pattern tree that specifies a set of constraints. The TPM problem is to find the nodes in the XML tree that satisfy all the constraints.

Previous research on the evaluation and optimization of path expressions fall into two classes. *Navigational* approaches traverse the tree structure and test whether a tree node satisfies the constraints specified by the path expression [23]. *Join-based* approaches first select a list of XML tree nodes that satisfy the node-associated constraints for each pattern tree node, and then pairwise join the lists based on their structural relationships (e.g., parent-child, ancestor-descendant, etc.) [28, 2, 5, 22]. Using proper labeling techniques [9, 8, 24], TPM can be evaluated reasonably efficiently by various join techniques (merge join [28], stack-based structural join [2], and holistic twig joins [5]).

Compared to the navigational techniques, join-based approaches are more scalable and enjoy optimization techniques from the relational database technology. However, there are inevitable difficulties:

1. Since choosing the optimal structural join order is NP-hard, the query optimizer relies heavily on heuristics [26]. When the query size (number of element sets generated in the first step) is reasonably large (say about 10), the optimization time may dominate the execution time. Thus, it is hard for the query optimizer to compromise between optimization and execution.
2. The selection-then-join methodology is not adaptive to the streaming XML data (e.g., SAX events) where the input streams could be considered as infinite and selection on the infinite input will not terminate.

In this paper, we propose a novel approach by combining the advantages of both navigational and join-based approaches. The rationale is based on the observation that some of the structural relationships imply *higher degree of locality* in the XML document than others, and thus may be evaluated more efficiently using the navigational approach. On the other hand, others represent more *global* relationships, and thus may be evaluated more efficiently using the join-based approach. For example, parent-child is a closer relationship than ancestor-descendant since finding the parent or child of a node requires only one navigation along the edge, but finding ancestor or descendant requires traversing a path or the whole subtree. Approaches that map XML documents and queries using interval encoding [10, 2] do not take advantage of this fact since they shred XML documents into small pieces (elements) and store them without considering their structural relationships. If we *cluster* XML elements at the physical

* This research was done while the author was visiting the University of Waterloo.

level based on one of the "local" structural relationships (say parent-child), the evaluation of a subset of the path expression consisting of only those local structural relationships can be performed more efficiently using a navigational technique without the need for structural joins.

Based on this idea, we define the *next-of-kin* (NoK) pattern tree to be a special pattern tree in which nodes are connected by parent-child and following-/preceeding-sibling relationships only (we call these *local relationships*). It is straightforward to partition a general pattern tree into NoK pattern trees, which are interconnected by arcs labeled with `//` or other "global" structural relationships such as following/preceeding. Given a general path expression, we first partition the pattern tree into interconnected NoK pattern trees, to which we apply the more efficient navigational pattern matching algorithm. Then, we join the results of the NoK pattern matching based on their structural relationships, just as in the join-based approach.

The effectiveness of this approach depends on the answers to the following two questions: (1) How many local relationships are there compared to global relationships in the actual queries? (2) How to efficiently evaluate NoK pattern matching so that its performance is comparable to or better than structural joins? The first question is hard to answer since it depends on the actual usage domain of the query, but a simple statistical analysis of the queries in the XQuery Use Cases [7] reveals that approximately $2/3$ of structural relationships are `/`'s, and $1/3$ are `//`'s [29]. This fact partly justifies that using NoK pattern matching in the first step will significantly reduce the number of structural joins in the second step.

To answer the second question, we conjecture that the efficiency of the NoK pattern matching algorithm relies on how well the physical storage scheme satisfies the clustering criteria. To justify this conjecture, we propose a simple and succinct physical storage scheme that not only supports efficient navigational NoK pattern matching, but also provides easy conditions (similar to the interval containment condition in the interval encoding approach) for subsequent global structural joins. Since the storage scheme has the locality property, an update of the XML document (insertion/deletion of an element) only affects part of the whole structure, making it more amenable to update than other techniques (e.g., the interval encoding [10]). Interestingly, it turns out that the physical storage scheme is analogous to the SAX stream format, so it follows that our matching algorithm on the physical storage scheme can be adapted to querying over streaming XML data as well.

In summary, our contributions in this paper are as follows:

- We propose a novel approach for matching a special type of pattern tree (NoK pattern tree) that only needs a single scan of the XML data in the *worst case*, and requires very small amount of main memory. The properties of single-pass and small footprint of the NoK pattern matching algorithm are also crucial to streaming XML processing.
- We design a novel, succinct physical storage scheme for XML documents that supports efficient NoK query evaluation. Joining answers of NoK pattern matching can also be efficiently evaluated in our storage scheme. Moreover, the locality preservation property makes the storage scheme more suitable for update.
- We compare the performance of our NoK algorithm based on our physical storage system with an existing interval encoding merge-join-based prototype (DI) [10], a holistic twig join algorithm [5], and a state-of-the-art native XML database system X-Hive/DB.The results indicate that our system outperforms DI in all cases, and holistic twig join and X-Hive in most cases.

The rest of the paper is organized as follows: in Section 2, we briefly introduce the basic notations and definitions used in the rest of the paper. In Section 3, we present the algorithm for NoK pattern matching at a logical level. In Section 4, we present the design of our physical storage scheme. In Section 5, we introduce how to apply the logical level NoK pattern matching algorithm to our physical storage. Section 6 presents the implementation and our experimental results. In Section 7 we discuss related work and compare them with ours. Finally, we conclude in Section 8.

2. Preliminaries

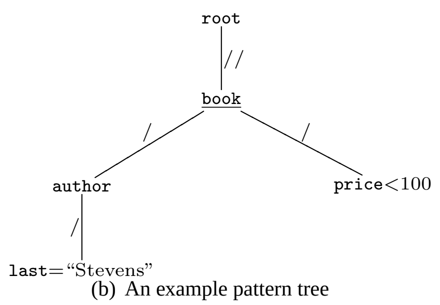
We briefly illustrate the XML data model and the pattern tree formalism by the following example, which we use throughout the paper. More details about XQuery path expressions can be found in [4, 6].

Example 1 Consider the bibliography XML document, in Figure 1(a), excerpted from the XQuery Use Cases [7]. The query "find all books written by Stevens whose price is less than 100" can be answered by the following path expression `//book[author/last="Stevens"][price<100]`. A *pattern tree* (which is defined shortly) that represent this path expression is shown in Figure 1(b). □

Each XML document can be modeled as a tree where the subelement relationship in the XML file corresponds to the child relationship in the tree. To efficiently store the tree, we need a mapping from the tag (element) names to the characters in an alphabet Σ—the short representations of tag names. For example, one possible mapping of tag names in Figure 1(a) to the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{i}, \texttt{j}, \texttt{z}\}$ could be as follows:

```
<bib>
 <book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison−Wesley</publisher>
  <price>65.95</price>
 </book>
 <book year="1992">
  <title>Advanced Programming in the Unix Environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison−Wesley</publisher>
  <price>65.95</price>
 </book>
 <book year="2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</lst><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
 </book>
 <book year="1999">
  <title>The Economics of Technology and Content for Digital TV</title>
  <editor>
   <last>Gerbarg</last><first>Darcy</first>
   <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
 </book>
</bib>
```

(a) An XML bibliography file



(b) An example pattern tree

Figure 1. An XML file and a pattern tree

bib → a	book → b	@year → z
author → c	title → e	publisher → i
price → j	first → f	last → g

Following this mapping, the XML document in Figure 1(a) can be represented as a tree, which we call the *subject tree* or *XML tree* (Figure 2). In the subject tree, only tag names and their structural relationships are preserved. The value of each tree node is detached from the structure and stored separately. We shall discuss the reason in Section 4.

A *pattern tree*, which we briefly introduced in the previous section, is a graphical representation of constraints specified in a path expression. A path expression can specify three types of constraints: tag-name constraints, value constraints, and structural relationship constraints. For example, the path expression in Example 1 specifies the constraints in the following formula:

$$\begin{aligned}\{\ b \mid\ & \mathrm{tag}(b) = \text{“book”} \ \wedge\ \exists a, l, p\ \mathrm{tag}(a) = \text{“author”} \wedge \\ & \mathrm{tag}(l) = \text{“last”} \ \wedge\ \mathrm{tag}(p) = \text{“price”} \wedge \\ & \mathrm{value}(l) = \text{“Stevens”} \ \wedge\ \mathrm{value}(p) < 100 \wedge \\ & \mathrm{descendant}(\mathrm{root}, b) \ \wedge\ \mathrm{child}(b, a) \ \wedge\ \mathrm{child}(a, l) \wedge \\ & \mathrm{child}(b, p)\}\end{aligned}$$

where $\mathrm{tag}()$ defines the tag-name constraint, $\mathrm{value}()$ defines the value constraint, and $\mathrm{child}()$ and $\mathrm{descendant}()$ define the structural relationship constraints. Given a pattern tree and a subject tree, the pattern matching problem is to find the set of subsets of subject tree nodes such that each subset satisfies all the constraints.

Figure 1(b) is the pattern tree representing the above constraints. Nodes in the pattern tree represent tag-name and value constraints (`root` is a special node representing the root of the XML tree), and edge labels represent structural relationship constraints between nodes. The underlined node (book) is the *returning node*, which means that nodes in the subject tree that match this returning node should be returned as the result of pattern matching. We assume that there is only one returning node in this paper. However, our algorithms can be easily extended to support multiple returning nodes by associating each of them a resulting list of nodes. It can be proven that any axis in XPath can be converted to a subtree consisting of edges whose labels are in the set $\{., /, //, \blacktriangleleft\}$, which correspond to self, child, descendant, following axes, respectively [29].

A *next-of-kin (NoK) pattern tree* is a special pattern tree that consists of edges whose labels are in $\{/, \lhd\}$, where $\lhd$ represents following-sibling axis. The NoK pattern tree is a tree if we only consider / arcs. Allowing $\lhd$ arcs makes it a layered graph—the children of each node of the NoK pattern tree is a directed acyclic graph (DAG) (We shall stick with the name NoK pattern tree for convenience even though different types of arcs form a graph.). Any pattern tree can be partitioned into NoK pattern trees that are connected by // arcs and ◀ arcs [29]. Therefore, we only need to develop a pattern matching algorithm for NoK pattern trees and then join the partial results of NoK pattern matching using structural joins based on // or ◀ relationships.

3. NoK pattern matching at the logical level

There are two steps in the process of matching NoK pattern trees to the subject tree: locating the nodes in the subject tree to start pattern matching, and NoK pattern match-

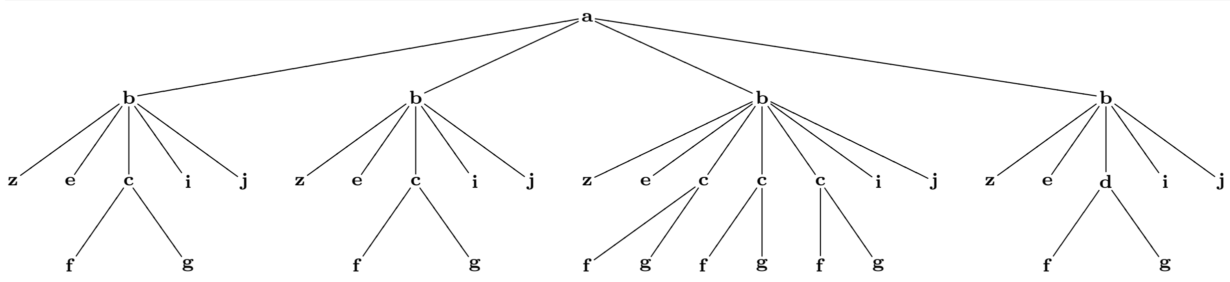


Figure 2. Subject tree representation of the bibliography XML document

ing from that starting node. The first step is needed since a NoK pattern tree `b/c` could be obtained from the path expression `/a//b/c` in which case any descendant of `/a` could be a starting point for NoK pattern matching.

In string pattern matching, the major concern is how to efficiently locate the starting points, while matching the string itself is straightforward. In the case of NoK pattern matching, both steps are nontrivial. There could be many options to locate the starting point:

Naïve approach: Traverse the whole subject tree in document order and try to match each node with the root of the NoK pattern tree. If a matching node is found, then start the NoK pattern tree matching process from that node. This is exactly what might be done in the streaming XML context.

Index on tag names: If we have a B+ tree on tag names, an index lookup for the root of the NoK pattern tree will generate all possible starting points.

Index on data values: If there are value constraints in the NoK pattern tree (such as `last`="Stevens" in Figure 1(b)), and we have a B+ tree for all values in the XML document, we can use that value-based index to locate all nodes having the particular value and use them as the starting points.

In our experiments, we implemented all three strategies and investigated their performance difference.

Having established the starting points, NoK pattern matching needs to deal with the unordered nature of siblings. That brings up the complexity that there could be more than one pattern tree node that matches a subject tree node. Moreover, we need to deal with the partial order constraints on siblings specified by the following/preceeding-sibling axes (recall that, in general, the children of a pattern tree node is a DAG connected by $\lhd$ arcs). We call the children of a *pattern* tree node *frontiers* if their sibling-indegree is 0, i.e., no sibling occurs before them according to the following/preceeding-sibling constraints. The frontiers represent the current ready-to–match nodes, and the set should be dynamically maintained since a matched frontier should be deleted (if it is not the returning node) and its "following siblings" in the pattern tree should be added if their sibling-indegree is now zero. This process is codified in Algorithm 1, which is a logical-level NoK tree pattern matching algorithm that returns TRUE if the pattern tree rooted at $proot$ matches the subject subtree rooted at $snode$ (the starting node) in the subject tree. Initially, the third parameter R is set to $\emptyset$, and it will contain a list of subject tree nodes (in document order) that match the returning node. We also assume that the label of $proot$ matches that of $snode$.

In lines 1–2 of Algorithm 1, if $proot$ is found to be the returning node in the pattern tree, its matching $snode$ is put in the result list R. Since there could be multiple subject tree nodes that match the returning node in different recursive calls, $snode$ must be appended to the resulting list. Lines 4 and 13 contain the only two operations on the subject tree. Together they implement the traversal of all children of $snode$ from left to right. During the traversal, if a subject tree node u matches a frontier node

Algorithm 1 NoK Pattern Matching

```
NPM($proot, snode, R$)
 1: if $proot$ is the returning node
 2:     then LIST-APPEND($R, snode$);
 3: $S \leftarrow$ all frontier children of $proot$;
 4: $u \leftarrow$ FIRST-CHILD($snode$);
 5: repeat
 6:         for each $s \in S$ that matches $u$ with both tag
              name and value constraints
 7:             do
 8:                 $b \leftarrow$ NPM($s, u, R$);
 9:                 if $b =$ TRUE
10:                   then $S \leftarrow S \setminus \{s\}$;
11:                        delete $s$ and its incident arcs
                             from the pattern tree;
12:                        insert new frontiers caused by
                             deleting $s$;
13:         $u \leftarrow$ FOLLOWING-SIBLING($u$);
14:     until $u =$ NIL or $S = \emptyset$
15: if $S \neq \emptyset$
16:     then $R \leftarrow \emptyset$;
17:         return FALSE;
18: return TRUE;
```

s, satisfying both tag-name and value constraints, we recursively match the subtrees rooted at u and s (line 6–8). If the whole subtrees match, s should not be considered as a candidate for matching subsequent subject tree nodes and its following-siblings in the pattern tree should be inserted into the frontier set if they qualify when deleting s and its incident arcs (line 9–12). The rest of the pseudocode cleans up the resulting list R if only part of the pattern tree was matched when traversing the children of $snode$ is exhausted—FOLLOWING-SIBLING returns NIL in line 13.

Note that Algorithm 1 accesses subject tree nodes in a depth-first manner. This means that subject tree nodes are accessed in the document order. This property is crucial to the proof of Proposition 1 given in Section 5.

Example 2 Consider the subject tree in Figure 2 and the NoK pattern tree in Figure 1(b) with tag names properly translated (`b[c/g="Stevens"][j<100]`). Suppose the starting point $snode$ is the first `b` in the subject tree, which matches the $proot$ and is appended to R, the algorithm iterates over `b`'s children to check whether they match with any node in the the frontier set {`c`,`j`}. When the third child of $snode$ matches with `c`, a recursive call is invoked to match the NoK pattern `c/g="Stevens"` with the subtree rooted at $snode/c$. When the recursive call returns TRUE, the algorithm continues to check the other children and eventually `j` is matched, causing the frontier set to be $\emptyset$. After that, the result R contains the starting point `b`. □

It is clear from the algorithm that every $snode$'s child will be visited exactly once, but in some special cases, its grandchildren (and great-grandchildren and so on) could be visited multiple times through multiple recursive calls. For example, in `/a[b/c][b/d]`, `a` has two children `b`'s and they should be both in the frontier when `a` is matched with $snode$. Since every $snode/b$ node matches both `b`'s in the frontier, two recursive calls will be invoked to match the two branches (`b/c` and `b/d`), so every grandchild of $snode$ will be visited exactly twice for matching `c` and `d`. In the worst case, there will be $|S|$ recursive calls at each level (when all frontiers nodes match with the current node of subject tree). Assume there are l levels in the pattern tree, s_i and p_i denote the number of nodes at level i in the subject tree and pattern tree, respectively, the maximum number of recursive calls at each level will be $O(s_i \cdot p_i)$. The complexity the whole algorithm is simply the sum of the number of recursive calls at each level $\sum_{i=1}^{l} O(s_i \cdot p_i) = O(mn)$, where $\sum_{i=1}^{l} s_i = m$ and $\sum_{i=1}^{l} p_i = n$, and m and n are the number of nodes in the pattern tree and subject tree, respectively.

4. Physical storage

Our desiderata for designing the physical storage scheme are as follows:

1. The XML structural information (subject tree) should be stored separately from the value information. The reason for this is explained in Section 4.1.
2. The subject tree should be "materialized" to fit into the paged I/O model. By materialization, we mean the two-dimensional tree structure should be represented by a one-dimensional "string". The materialized string representation should be as succinct as possible, yet still maintain enough information for reconstructing the tree structure. The justification for this is given in Section 4.2.
3. The storage scheme should have enough auxiliary information (e.g., indexes on values and tag names) to speed up NoK pattern matching.
4. The storage scheme should be adaptable to support updates.

In the subsequent two subsections, we shall introduce how we manage the value information and structural information, respectively.

4.1. Value information storage

The first issue in the desiderata is based on two observations: Firstly, an XML document is a mixture of schema information (tag names and their structural relationships) and value information (element contents). The irregularity of contents (variability of lengths) makes it hard (inefficient) for the query engine to search for certain schema/content information. Secondly, any path query can be divided into two subqueries: pattern matching on the tree structure and selection based on values. For example, the structural constraints and value constraints in the path expression in Figure 1(b) are `//book[author/last][price]` and `last="Stevens"` $\wedge$ `price<100`, respectively. The final result could be joined by the results returned by the subqueries. Separating the structural information and the value information allows us to separate the different concerns and address each appropriately. For example, we could build a B+ tree on the value information and a path index (suffix tree, for example) on the structural information without worrying about the other part.

After the separation, we need to somehow maintain the connection between structural information and value information. We use Dewey ID [14] as the key of tree nodes to reconnect the two parts of information, e.g., the Dewey IDs of the root `a` and its second child `b` are 0, and 0.2, and so on. The reason that we use Dewey ID instead of giving each node a permanent ID is that Dewey ID contains the structure information and can be derived automatically during the tree traversal. That eliminates the need to keep the ID information in the tree structure (cf. Section 4.2). Given a Dewey ID, we need another B+ tree to quickly locate the

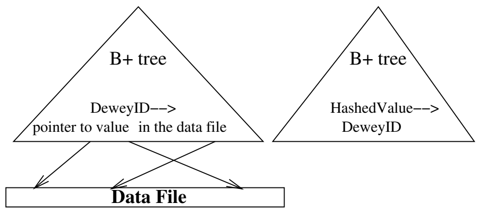


Figure 3. Data file and auxiliary indexes

value of the node in the data file. This data file and its auxiliary data are shown in Figure 3.

The value information for all subject tree nodes is stored sequentially in a *data file* or it can be retained in the original XML document. To evaluate the value-based constraints efficiently, we build a B+ tree on the data file whose keys are the *hashed* data values and returns a set of Dewey ID's whose nodes contain that value. The purpose of the hash function here is to map any data value (could be variable length string) to an integer that can be compared quickly. Different values that are hashed to the same key can be distinguished by looking up the data file directly. Careful selection of the hash function would significantly reduce this type of conflicts.

Example 3 In the data file, each element content could be represented by a binary tuple $(len, value)$, where len is the length of the value. The value information for the XML document in Figure 1(a) can be organized as a list of records: (4,"1994"), (18,"TCP/IP Illustrated"), (14,"Addison-Wesley"), (7,"Stevens"), (5,"65.95"), and so on. The position of these records in the data file are kept in the Dewey ID B+ tree. If there are more than one node with the same value, we can keep only one copy and let these nodes point to the same position in the data file. □

If the XML file is updated, the value can be easily appended to the end of the data file. However, both indexes need to be updated. The value-based B+ tree can be updated incrementally based on insertion/deletion of keys. Due to the nature of Dewey IDs, the node ID B+ tree may need to be reconstructed if many IDs have been updated.

4.2. Structural information storage

One way to materialize the tree is to store the nodes in pre-order and keep the tree structure by properly inserting pairs of parentheses as introduced in [17]. For example, `(a(b)(c))` is a string representation of the tree that has a root `a` and two children `b` and `c`. The "`(`" preceding `a` indicates the beginning of a subtree rooted at `a`; its corresponding "`)`" indicates the end of the subtree. It is straightforward that such string representation contains enough information to reconstruct the tree. However, it is not a succinct representation because each node (a character in the string) actually implies an open parenthesis. Therefore, we can safely remove all open parentheses and only retain closing parentheses as in `a b)c))`. Note that this representation can be further compressed by replacing any series of closing parentheses with a number indicating how many of those closing parentheses there are. However, this introduces the difficulty that we do not know how many bits are needed for encoding the number, unless we parse the XML document beforehand. However, parsing beforehand is impossible in the context of streaming XML where we have no knowledge of the upcoming events (closing tag or deeper nesting). Thus we keep the closing parentheses "`)`".

Example 4 Figure 4 shows the string representations of the subject tree in Figure 2 (At the physical level, the pointers in the figure are not stored. They only serve to easily identify the end of a subtree to the reader.). If the string is too long to fit in one page, it can be broken up into substrings at any point and stored in different pages. Assume each character in Σ is 2 bytes long, "`)`" is 1 byte long, and each page is 20 bytes long (the number is chosen for illustration only), the string can be divided into six pages separated by the dashed lines in the figure. □

To speed up the query process, we need to store extra information in each page. The most useful information for locating children, siblings and parent is the node level information, i.e., the depth of the node from the root. For example, assuming the level of the root is 1 in Figure 4, the level information for each node is represented by a point in the 2-D space under the string representation (the x-axis represents nodes and the y-axis represents level). For each page, an extra tuple $(\mathtt{st}, \mathtt{lo}, \mathtt{hi})$ is stored, where `st` is the level of the last node in the previous page, `lo` and `hi` are the minimum and maximum levels of all nodes in that page, respectively (Note that `st` could be outside the range $[\mathtt{lo}, \mathtt{hi}]$.). This tuple can be thought of as a feather-weight index for guessing the page where the following sibling or parent is located. We shall introduce its usage in Section 5.

Note that when streaming XML (e.g., SAX events) are parsed so that every open tag of an element is translated to a character in Σ and every closing tag is translated to a "`)`", the result is exactly the same as our physical string representation. Therefore, our single-pass NoK pattern matching algorithm (presented in Section 5) based on this physical string representation can be adapted to the streaming XML context, except that page headers (which help to skip page I/O's) are not necessary in the streaming context since each page needs to be read into main memory anyway.

In addition to these advantages, it is also fairly easy to insert and delete nodes from the string representation of the tree. Attaching a subtree to a leaf can be done by inserting the string representation of the subtree between the left

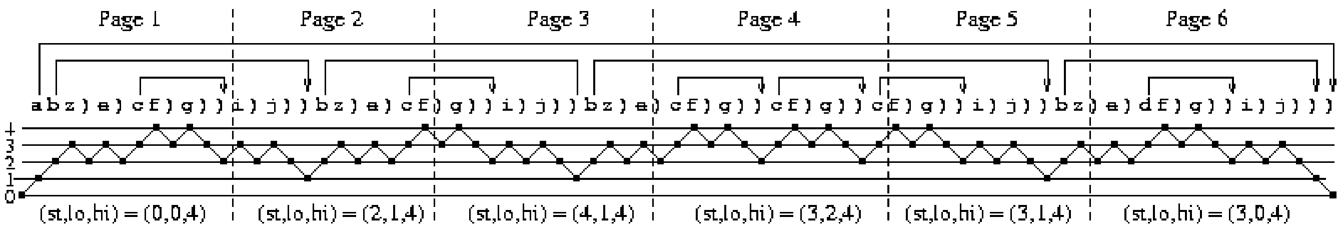
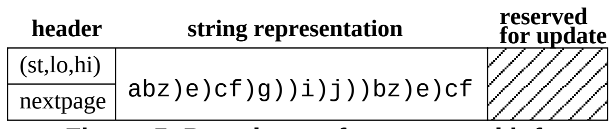


Figure 4. The string representation of an XML tree

character and it's corresponding "`)`". For example, to insert `ab)c))` as a subtree of the first `f` node in page 1, we can allocate a new page with the content `ab)c))`, cut-and–paste the content after `f` in page 1 to the end of content of the new page, and insert the new page into the page link between pages 1 and 2. The $(\mathtt{st}, \mathtt{lo}, \mathtt{hi})$ information for page 1 should be changed accordingly. Inserting a subtree to a non-leaf node is slightly more complicated. For example, if `a` is inserted in between the root `a` and its second child `b`, this requires the insertion of an additional "`)`" after the rightmost descendant of `b`. This can be handled by controlling the load factor of each page, thereby reserving some of the page for insertion and by keeping a next page pointer in the header in case a new page is inserted. The page layout is shown in Figure 5.

According to the page layout, the number of nodes in each page can be calculated easily: assume that each page is 4KB, of which 20% of the space is reserved for update; each character in Σ is 2 bytes long and "`)`" is 1 byte long. Then each node occupies 3 bytes (because each node consists of a character in Σ and a "`)`" character); each parameter in the vector $(\mathtt{st}, \mathtt{lo}, \mathtt{hi})$ occupies 1 byte, and the page index occupies 4 bytes. Consequently, the number of nodes in a page is around 1000. We call this number the *capacity*, $\mathcal{C}$, of the page. It can be calculated by the formula: $\mathcal{C} = \frac{B\times(1-r)-V-I}{S+P}$, where B is the page size, r is the ratio for space reserved for update, V is the size of vector $(\mathtt{st}, \mathtt{lo}, \mathtt{hi})$, I is the page index length, S is the length of character in Σ, and P is the length for encoding of "`)`". As we calculated above, the value of $\mathcal{C}$ is around 1000 to 3000 by substituting reasonable values to these parameters. Our experiments show that the string representation of the tree structure is only about $1/20$ to $1/100$ of the size of the XML document.

Now assume that the subject tree has 10 billion nodes (the size of the original XML document is about 200GB to 1TB according to our statistics), then we need about 3 to 10 million pages to store the string representation of the tree structure. If we load the page headers (assuming each is 7 bytes long) to main memory, we only need 21MB to 70MB. In modern computer systems, this is *really* small for handling up to 1 terabyte of data.



header	string representation	reserved for update
(st,lo,hi)	`abz)e)cf)g))i)j))bz)e)cf`	▨
nextpage		

Figure 5. Page layout for structural info.

Then the natural question is that if we load the header information into main memory beforehand, how does it help in speeding up the path queries? We shall answer this question in the next section.

5. XML path queries at the physical level

In the NoK pattern matching algorithm (Algorithm 1), the only operation on the subject tree is the iteration over children of a specific node in their document order. Using the physical storage technique proposed in the previous section, this operation is divided into two primitive operations: FIRST-CHILD to find the first child of a node, and FOLLOWING-SIBLING to find the following sibling of a node. The physical level NoK pattern matching algorithm simply uses the physical level FIRST-CHILD and FOLLOWING-SIBLING operations to perform the iteration in lines 4 and 13 in Algorithm 1.

According to the pre-order property of the string representation, these two operations can be performed by looking at the node level information of each page from left to right without reconstructing the tree structure. The basic idea is illustrated in the following example.

Example 5 Consider the string representation in Figure 4. Suppose we want to find the first child of character `b` in the first page. Since the nodes are pre-ordered, the first child of `b` must be the next character if it's not a "`)`". This condition is equivalent to saying that the first child of a node at level l is the next character if its level is $l + 1$. In Figure 4 the answer is `b`'s immediate right neighbor `z`.

Now, suppose we want to find `b`'s following sibling. Again, since the nodes are pre-ordered, the following sibling must be located to the right of `b` in the string and its level must be the same. Moreover, the target character must not to be too far to the right since, in this case, it could be `b`'s cousin (share the same grandparent but not the parent). Therefore, there must be another constraint: no intermediate character (i.e., cousin) whose level is 2 less than b's level should be in the string between b and b's following sibling. In Figure 4, the answer is b in page 2, but there is no following sibling of j in the second page. □

Given a page, it is straightforward to calculate the level information for each node: initially the level is set to st in the page header (st in the first page is always 0), scan the string from left to right, if the character is in Σ, its level is incremented by 1, otherwise (i.e., a closing parenthesis), its level information is decremented by 1. For example, the levels for the nodes in the first page are 0123232343432.

Algorithm 2 gives a straightforward implementation of the FIRST-CHILD and FOLLOWING-SIBLING operations.

The READ-PAGE subroutine reads a page from disk to main memory and calculates the level information described above. It takes the page number p as input parameter, and returns the page content and level information to the next two parameters $\mathbf{A}$ and $\mathbf{L}$, which are two-dimensional arrays, where $\mathbf{A}[p]$ and $\mathbf{L}[p]$ are strings (e.g., abz)e)cf)g)) and 0123232343432 for page 1) representing the content and level information of page p, respectively.

The FIRST-CHILD and FOLLOWING-SIBLING subroutines call READ-PAGE to read a page and calculate the level information when necessary. They take two parameters p and o that are the page number and the offset in the page, respectively. They check the string representation and level information stored in $\mathbf{A}$ and $\mathbf{L}$ and return a character representing the tag name of first child or following sibling.

The I/O complexity of the FIRST-CHILD is straightforward: two page I/O's in the worst case to get the next character in the string. The FOLLOWING-SIBLING operation may scan the whole file before finding the next character with the same level information. In fact, this is the case for finding the following sibling of root a in Figure 4. To avoid unnecessary page I/O's, we should exploit the maximum and minimum level information in each page as described in the page header. The idea is based on the fact that if the current node u with level l has a following sibling, the page that contains this following sibling must have a character ")" with level $l-1$ (this is the closing parenthesis corresponding to u). If $l-1$ is not in the range $[\texttt{lo}, \texttt{hi}]$ of a page, it is clear that this page should not be loaded. As we described in the previous section, we can keep all the page headers in main memory with very low cost, and greatly reduce the number of page I/O's. In the case of locating a's following sibling, only two page I/O's are needed (pages 1 and 6). This optimization can be easily implemented by modifying the READ-PAGE subroutine in Algorithm 2, so we do not give the actual algorithm here.

The FIRST-CHILD and FOLLOWING-SIBLING subroutines correspond to the child and following-sibling axes in a path expression. Other axes (e.g., parent, // and following) can be easily composed by using these two operations. For example, given a node u in the string representation, its descendants are those characters located in between u and its following sibling (more precisely it should be all characters in between u and its first right-side character whose level is $level(u) - 1$). This implies that the interval $\langle p_1 * \mathcal{C} + o_1, p_2 * \mathcal{C} + o_2 \rangle$, where p_1, p_2, c_1, c_2 are the page number (p_i) and offset (c_i) of a character and its corresponding ")", respectively, can be used in the condition for structural joins just as in the interval encoding approach.

Algorithm 2 Primitive Tree Operations

```
READ-PAGE($p, \mathbf{A}, \mathbf{L}$)
1:  if page $p$ is invalid
2:      then return FALSE;
3:  if page $p$ is not in main memory
4:      then read page $p$ in array $\mathbf{A}[p]$;
5:          calculate level array $\mathbf{L}[p]$ for page $p$;
6:  return TRUE;
FIRST-CHILD($p, o$)
1:  if READ-PAGE($p, \mathbf{A}, \mathbf{L}$) = FALSE
2:      then return NIL;
3:  if $o = \mathbf{A}.len$
4:      then return FIRST-CHILD($p + 1, 0$);
5:  elseif $\mathbf{L}[p][o+1] = \mathbf{L}[p][o] + 1$
6:      then return $\mathbf{A}[p][o+1]$;
7:  else return NIL;
FOLLOWING-SIBLING($p, o$)
1:  $l \leftarrow \mathbf{L}[p][o]$;
2:  $j \leftarrow o + 1$;
3:  while READ-PAGE($p, \mathbf{A}, \mathbf{L}$) = TRUE
4:      do
5:          while $j < \mathbf{A}.len$
6:              do
7:                  if $\mathbf{L}[p][j] = l - 2$
8:                      then return NIL;
9:                  elseif $\mathbf{L}[p][j] = l$ and $\mathbf{A}[p][j] \neq ')'$
10:                     then return $\mathbf{A}[p][j]$;
11:                 $j \leftarrow j + 1$;
12:         $p \leftarrow p + 1$;
13:         $j \leftarrow 0$;
14: return NIL;
```

Proposition 1 *Given a string representation of the subject tree $\mathcal{S}$ and a NoK pattern tree $\mathcal{P}$, suppose the maximum number of descendants of the second level nodes (e.g., nodes labeled with b in Figure 2) in $\mathcal{S}$ is n. The physical level NoK pattern matching algorithm reads every page at most once (single-pass), and requires only $n/\mathcal{C}$ pages in main memory (recall $\mathcal{C}$ is the capacity of the page).*

PROOF From the analysis of Algorithm 1, we know that in a special case, the algorithm might access a subject tree node u more than once if $level(u) > 2$. In our physical storage scheme, the descendants of u are stored before its following sibling. Since Algorithm 1 matches subject tree nodes in a depth-first manner (matches all of u's descendants first before following sibling and never reads back), in the worst case we only need to read all the pages that contain u's descendants in main memory, which requires a buffer size of $n/\mathcal{C}$ pages, and match them against all pattern tree branches. After the FOLLOWING-SIBLING is called, this buffer can be freed and those pages are read only once.$\blacksquare$

Since usually XML files are flat and the page capacity is around 1000, the number of pages needed in main memory is small in practice. We leave the development of a space and time optimal algorithm as future work.

6. Experimental evaluation

To assess the effectiveness of the proposed approach, we conducted extensive experiments and compared them with the performance of existing systems or prototypes that are based on interval encoding or other native physical storage schemes. Both the data and the queries are classified into categories so that we can test the efficiency of all approaches in different environments.

6.1. Experimental setting

We implemented the algorithms and physical storage prototype in Java with JDK 1.4. All the experiments were conducted on a PC with Pentium III 997MHz CPU, 512MB RAM, and 40GB hard disk running Windows XP.

We conducted our experiments using both synthetic and real data sets. The synthetic data sets (author, address, and catalog) are selected from XBench benchmark [27] in the data-centric category. The real data sets (Treebank and dblp) are selected from University of Washington XML Data Repository [1]. These data files are selected because they are either bushy (author, address, dblp) or deep (catalog, Treebank). Table 1 shows the statistical information of the the data sets and B+ tree indexes we built for them, in which $\mathbf{tree}, \mathbf{B+_t}, \mathbf{B+_v}, \mathbf{B+_i}$ denote the string representation of the tree structure, the B+ trees for tag names, values, and Dewey IDs, respectively.

Queries were carefully selected for the experiments to cover different aspects of path queries on the XML data. Our selection is based on the following three properties of path expressions:

Selectivity: A path expression returning a small number of results should be evaluated faster than those returning a large number of results. To evaluate whether our algorithm is sensitive to selectivity, we divided queries into three categories based on their selectivity: high (several results), moderate (greater than 10 but less than 100 results), and low (greater than 100 results).

Query	Category	Example query
Q1	hpy	`/a/b[c="hi"]`
Q2	hpn	`/a/b/c/d`
Q3	hby	`/a/b[c="hi"][d="hi"]/e`
Q4	hbn	`/a/b[c][d][e][f]`
Q5	mpy	`/a/b[z="mod"]/d/e`
Q6	mpn	`/a/b/e`
Q7	mby	`/a/b[c="mod"][d="mod"]`
Q8	mbn	`/a/b[c][d][e]`
Q9	lpy	`/a/b[c="low"]/d`
Q10	lpn	`/a/b/c`
Q11	lby	`/a/b[c="low"][d="low"]`
Q12	lbn	`/a/b[c][d]`

Table 2. Query categories

Topology: The shape of the pattern tree could be a single-path or bushy (two or more leaf nodes) and may contain `//`-arcs. Some systems may have different performance in these cases, but the I/O cost of our algorithm should be the same, except that the main memory operations in the bushy case could be greater. We would like to experimentally verify the analytical results.

Value constraints: The existence of value constraints and index on values may be used for fast locating the starting point for NoK pattern matching, especially when the selectivity is high. Therefore, queries having value constraints may be used to justify the effectiveness of value-based indexes.

Combining these three criteria, we designed queries in twelve categories shown in Table 2. Each category is denoted by a string of length three, where each position denotes one of the above criterion. The character in each position stands for: low (l), moderate (m), or high (h) for selectivity; path (p), or bushy (b) for topology; and yes (y), or no (n) for existence of value constraints. The tag names and constants in the example queries are dummy and they should be replaced by appropriate values in different test files. We also tested `//` axis by randomly substituting it for a `/` axis.

6.2. Performance evaluation and analysis

We tested our system against two join-based algorithms based on interval encoding—dynamic interval (DI) [10] and TwigStack [5], as well as a state-of-the-art native XML database system X-Hive/DB version 4.1.1. For each data set, we chose a representative path expression in each of the twelve categories. The performance evaluation results are shown in Table 3. Each running time is the average over three executions. Some categories are not applicable (denoted as "NA" in the table) to the data sets (e.g., author and

data set	**size**	**#nodes**	**avg. depth**	**max depth**	$\mathbf{|tags|}$	$\mathbf{|tree|}$	$\mathbf{|B+_t|}$	$\mathbf{|B+_v|}$	$\mathbf{|B+_i|}$
author	1.2 MB	$15,006$	3	3	8	0.035 MB	0.18 MB	0.33 MB	0.4 MB
address	17 MB	$403,201$	3	3	7	0.5 MB	5 MB	12 MB	11 MB
catalog	30 MB	$620,604$	5	8	51	1.2 MB	8 MB	15 MB	13 MB
TreeBank	82 MB	$2,437,666$	8	36	250	5.3 MB	58 MB	80 MB	72 MB
dblp	133 MB	$3,332,130$	3	6	35	8 MB	62 MB	150 MB	180 MB

Table 1. Statistic information of data sets

address data sets do not have moderate selectivity queries without value constraints), or the queries we selected for the category contain some functionalities that were not implemented (denoted as "NI" in the table) by a particular system (e.g., DI does not support value comparisons other than equality as of the date we perform the experimentation).

To prepare for the test, we created all the indexes (ID, tag-name, and value) for the data sets. To conduct fair comparisons, we also created indexes (ID, tag-name, and value) for X-Hive. We implemented the TwigStack algorithm in a way such that different tree nodes with different tag names are stored separately in a file sorted by document order. Each file contains the nodes constituting an input stream associated with a node in the twig. In order to speed up value comparisons, we also created a B+ tree for the value nodes. DI has only limited support for tag-name index at this time, so we did not use index on the tests for DI. This is one of the reasons that DI did not perform as well as other systems. We apply a very simple heuristic to index usage: whenever there are value constraints, we always use the value index to locate the starting point for NoK pattern matching. If there are more than one value constraints, the most selective one is used. If there are no value constraints, we pick the tag name which has the highest selectivity. If the selectivity is "high" (defined based on our heuristic), we use the tag-name index for locating starting points, otherwise we use a sequential scan. This heuristic allows us to test the effectiveness of value and tag-name indexes. Experiments show that sometimes value index is more effective than tag-name index (e.g., in Treebank high selective categories), and sometimes tag-name index is more effective (e.g., in catalog). This is because values in Treebank were randomly generated and has higher selectivity than tag names.

Another reason for the good performance of NoK (as well as TwigStack) is that it does not need to materialize intermediate results for multiple joins. Materializing intermediate results or recomputing partial results is inevitable in bushy path expressions for DI. For example, in the path expression `/a/b[c][d]/e`, element `b` needs to be tested for children `c` and `d`, and then return children `e`. Each of the three operations need a join with nodes returned by `/a/b`, while in a single-path query, DI could use a pipelined plan and avoid materialization. Therefore, DI is topology sensitive, but our system is not, as shown in Table 3.

Moreover, since both materialization and re-computation are expensive operations when the intermediate result is large, DI has to perform the same amount of work disregarding the result size, i.e., DI is not sensitive to the selectivity. Generally, the running time of our system decreases when the selectivity of the *starting points* increases. Our experiment shows that the selectivity of staring points can be a fairly good approximation for selectivity of final results if we choose the most selective index (value or tag-name-based) for locating the starting points.

In comparison to the TwigStack algorithm, our algorithm performs fewer fruitless scans because it does not need to traverse a subtree if its root does not match a pattern tree node. However, TwigStack has to scan all streams associated with leaf nodes in the pattern tree. Although XB-tree [5] or other index structures [16] might compensate for this problem, the storage basis (interval encoding) lacks the flexibility for update and for processing streaming XML.

In summary, our system is comparable to DI, TwigStack, and X-Hive in some cases and outperforms them in most cases. In particular, our system is sensitive to the selectivity, insensitive to the topology of pattern tree, and could take advantage of the existence of value constraints.

7. Related work and comparisons

In general, tree pattern matching can be classified into ordered tree pattern matching (OTPM) problem and unordered tree pattern matching (UTMP) problem depending on the ordering of siblings in the pattern tree. The OTPM problem can be solved very efficiently—$O(n\sqrt{m}$ polylog $m)$, where m and n are respectively the sizes of pattern tree and data tree, by using suffix trees or other data structures [15, 11]. However the pattern tree generated by path expressions are generally unordered since branchings in the pattern tree are caused by multiple predicates that are unordered. To be more precise, we shall consider partially ordered pattern tree since two nodes can be connected by the following-sibling or preceeding-sibling axes. The UTPM problem is generally tackled by the join-based approach introduced above. Early implementations follow the formal semantics and treat a path expression as a

file	systems	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
author	DI	0.25	0.24	0.6		0.24		NI		NI	0.29	NI	0.87
	X-Hive	0.34	0.28	0.45	NA	0.38	NA	0.48	NA	0.78	0.43	0.85	0.23
	TwigStack	0.25	0.21	0.28		0.34		1.5		1.6	0.38	2.3	0.48
	NoK	0.2	0.23	0.29		0.3		0.29		0.58	0.17	0.69	0.17
address	DI	2.53	2.6	6.6		2.49		8.9		2.57	2.73	NI	4.81
	X-Hive	0.3	1.7	0.47	NA	1.03	NA	0.72	NA	0.79	2.3	0.87	4.73
	TwigStack	0.3	0.31	0.26		1.2		1.74		1.79	1.9	101	4.2
	NoK	0.32	0.05	0.27		0.83		0.77		1.5	3.2	1.39	3.8
catalog	DI	8.8	3.25	14		9.4		NI		8.9	2.87	NI	13
	X-Hive	1.3	2.16	0.67	NA	1.4	NA	1.4	NA	1.7	4.9	1.6	5.4
	TwigStack	1.01	1.2	0.81		2.01		1.3		0.8	0.9	2.4	1.1
	NoK	0.37	0.1	0.43		1.2		1.4		2.5	1.1	2.4	1.3
Treebank	DI	26.3	11.7	45.5	13		27.4		43		14.2		43.6
	X-Hive	0.61	8.2	0.6	7.2	NA	19.7	NA	14	NA	2.9	NA	14.2
	TwigStack	0.8	18.3	0.45	20.1		1.62		1.9		1.8		4.2
	NoK	0.35	0.77	0.37	0.74		0.51		0.65		2.1		1.2
dblp	DI	27	18.5	17.5	18.3	60.8	26	18.8	17.6	17.4	18	75.2	26.6
	X-Hive	0.97	19.4	12	11.36	10.8	8.6	12.9	8.85	2.78	16.2	9.7	10.7
	TwigStack	2.19	1.2	7.3	2.2	8.9	0.32	9.2	0.45	1.2	1.4	12.9	13.8
	NoK	0.6	0.1	2.8	3.2	1.22	0.57	10.8	0.5	1.5	0.89	0.62	0.99

Table 3. Running time (in sec) for DI, X-Hive, TwigStack, and NoK on different queries and data sets

sequence of steps, each of which takes input from the previous step and produces an output to the next step. This can be thought of as a special case of join-based approach that uses nested-loop join instead of merge join-like algorithms. Experimental results show that implementations following this approach suffer from exponential runtime in the size the of path expressions in the worst case [13]. That paper proves that path expressions can be answered in polynomial time, but their algorithm is based on main memory and cannot be adapted to streaming XML. Barton et al. [3] proposed a single-pass algorithm for streaming XML processing. They defined a subset of path expressions that only supports child, parent, ancestor and descendant axes, and no value constraints. Their algorithm is very similar to NoK pattern matching in the streaming XML context, but the latter is more efficient in the non-streaming context since it takes advantage of $\langle \mathtt{st}, \mathtt{lo}, \mathtt{hi} \rangle$ triples and value-based indexes to guide the navigation. Recently, Wang et al. [25] proposed an index that is efficient in matching pattern trees without using structural joins. However their index only works on ordered pattern tree. For unordered ones, they need to enumerate all possible ordered trees (which is exponential) and perform an index look-up to each of them.

On the XML physical storage part, a number of approaches have been proposed, including using flat file systems (e.g., Kweelt [20]), extending mature DBMS technologies such as relational DBMSs (e.g., IBM DB2, Oracle, and Microsoft SQL Server) or object-oriented DBMS (e.g., Ozone [19]), and building native XML repositories (e.g., Tamino [21], Natix [12], X-Hive, and Xyleme). Recently, Koch [18] proposed a physical storage scheme based on the binary representation of an unranked tree. The space it uses is comparable to ours but it requires more page I/O in the FOLLOWING-SIBLING operation if the tree is very deep since they do not maintain the level information. Interestingly, his approach can also be extended to the streaming XML context but requires two sequential scans. We use only one sequential scan in the worst case since NoK pattern tree is less expressive than the path expression he difined.

8. Conclusion and future work

In this paper, we have defined a special type of pattern tree—NoK pattern tree, and proposed a novel approach for efficiently evaluating path expressions by NoK pattern matching. The result of NoK pattern matching greatly reduces the number of structural joins in the later step. NoK pattern matching can be evaluated highly efficiently (only need a single scan of input data) using the physical storage scheme we proposed. Performance evaluation has shown that our system has better or comparable performance than the existing extended-relational (based on interval encoding) and native XML database management systems.

To further improve the performance, we can do more optimization on the locating step of NoK pattern tree matching process. For example, instead of linear scan, we can preprocess the NoK pattern tree in a similar fashion as Robin-Karp or Knuth-Morris-Pratt algorithms have done to string pattern matching. Another improvement is to use path index instead of tag-name index. This is particularly efficient when the selectivity of individual tag names are low but the selectivity of a path is high. Without exception, any storage scheme in the database systems should consider how to employ concurrency control and how it affect the update process. Although these are extremely important issues, we leave them as future work.

Acknowledgements Special thanks to David DeHaan and David Toman of University of Waterloo for providing support for testing against DI prototype. Research supported in part by a grant from Communications and Information Technology Ontario.

References

[1] UW XML Data Repository. Available at http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. 18th Int. Conf. on Data Engineering*, pages 141–152, 2002.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streming XPath Processing with Forward and Backword Axes. In *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language. Available at http://www.w3.org/TR/xquery/.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–322, 2002.

[6] D. Chamberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(40):597–615, 2002.

[7] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML Query Use Cases. Available at http://www.w3.org/TR/xmlquery-use-cases.

[8] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar. Labeling Your XML. preliminary version presented at CASCON'02, October 2002.

[9] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 271–281, 2002.

[10] D. DeHann, D. Toman, M. P. Consens, and M. T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 623–634, 2003.

[11] M. Dubiner, Z. Gallil, and E. Magen. Faster Tree Pattern Matching. *J. ACM*, 41(2):205–213, 1994.

[12] T. Fiebig, S. Helmer, C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. Technical report, University of Mannheim, 2002.

[13] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 95–106, 2002.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 16–27, 2003.

[15] C. M. Hoffmann and M. J. O'Donell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, 1982.

[16] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 273–284, 2003.

[17] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 3rd edition, 1997.

[18] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata -based Approach. In *Proc. 29th Int. Conf. on Very Large Data Bases*, 2003.

[19] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating Structured and Semistructured Data. *Lecture Notes in Computer Science*, 1949:297–323, 2000.

[20] A. Sahuguet. KWEELT, the Making-of: Mistakes Made and Lessons Learned. Technical report, University of Pennsylvania, November 2000.

[21] H. Schöning and J. Wäsch. Tamino - An Internet Database System. In *Advances in Database Technology — EDBT'00*, pages 383–387, 2000.

[22] D. Shasha, J. T. L. Wang, and R. Giugno. Algorthmics and Applications of Tree and Graph Searching. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 39–53, 2002.

[23] J. Simeon and M. Fernandez. Available at http://www-db-out.bell-labs.com/galax/.

[24] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 204–215, 2002.

[25] H. Wang, S. Park, W. Fan, and P. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 110–121, 2003.

[26] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[27] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. 20th Int. Conf. on Data Engineering*, 2004.

[28] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 425–436, 2001.

[29] N. Zhang and M. T. Özsu. Optimizing Correlated Path Expressions in XML Languages. Technical Report CS-2002-36, University of Waterloo, November 2002. Available at http://db.uwaterloo.ca/∼ddbms/publications/xml/TR-CS-2002-36.pdf.