

Faster Path Indexes for Search in XML Data

Nils Grimsmo*

Department of Computer and Information Science
Norwegian University of Science and Technology

Sem Sælands vei 7-9

7491 Trondheim

Norway

nilsgri@idi.ntnu.no

Abstract

This article describes how to implement efficient memory resident path indexes for semi-structured data. Two techniques are introduced, and they are shown to be significantly faster than previous methods when facing path queries using the descendant axis and wild-cards. The first is conceptually simple and combines inverted lists, selectivity estimation, hit expansion and brute force search. The second uses suffix trees with additional statistics and multiple entry points into the query. The entry points are partially evaluated in an order based on estimated cost until one of them is complete. Many path index implementations are tested, using paths generated both from statistical models and DTDs.

1 Introduction

With the advent of XML and query languages such as XPath and XQuery came the need for efficient ways to query the structure of XML documents. This article focuses on settings where a document collection can be indexed in advance, as opposed to querying on the fly. For efficient solutions to the latter problem, see for example Gottlob et al. (2005). An important component in many systems indexing XML is a *path index* (Bertino & Kim 1989) summarising and indexing all unique paths seen in the document collection. (Other names for similar structures are *representative objects* (Nestorov et al. 1997), *DataGuides* (Goldman & Widom 1997), and *access support relations* (Kemper & Moerkotte 1992).) For many document collections following schemas, this set of paths will be small compared to the total size of the data. The path index is in some way connected to a *value index* (or *content index*), which allows search for words or values.

XPath is a query language allowing search for *regular path expressions* in XML documents. It is a simple declarative language, but techniques used for XPath queries can also be components in more advanced procedural query languages such as XQuery. Many FLOWR expressions can also be rewritten to simpler XPath expressions (Michiels et al. 2007).

Assume the XML document shown in Figure 1,

*Supported by the Research Council of Norway under the grant NFR 162349.

Copyright ©2008, Australian Computer Society, Inc. This paper appeared at the Nineteenth Australasian Database Conference (ADC2008), Wollongong, Australia, January 2008. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 75, Alan Fekete and Xuemin Lin, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

and the XPath query `/a//c/"foo"`¹. There are two matches for the path part of the query, and four matches for the value predicate "foo", but only one match for the entire query, which is the third occurrence of "foo". Note that each unique path is only indexed once in a path index. There are two occurrences of the path `/a/b` in the example document, but there will only be one in a path index.

An efficient path index is important when indexing large heterogeneous document collections for structural querying. If the data has a very homogeneous structure, the number of unique paths seen is small, and the implementation of the path index itself is not significant. An example where document structure could be very heterogeneous, is an enterprise search engine, indexing all information generated by a business. This could be composed of the content from multiple databases, repositories of reports, etc.. Another case is search engines for the semantic web, where structural search is a key feature. It is hard to imagine a future for the semantic web without search engines that scale as well as current web search engines.

1.1 Previous approaches

Many approaches for indexing XML and supporting path queries have been proposed in the last ten years. They can mainly be divided into *node indexing*, *path indexing* and *sequence indexing*. In *node indexing*, one makes an inverted index for all XML tags, and one for all data values and terms. Given a simple path query, lookup all the tags and values, and merge the results. To be able to merge hits for XML elements, an encoding which tells whether a node is a child or descendant of another node is needed. Common solutions are the range based (*docid*, *start:end*, *level*) encoding (Zhang et al. 2001), the (*post*, *pre*) encoding (Grust 2002) and the prefix based *Dewey order* encoding (Tatarinov et al. 2002).

A "brute force" merge between element hits may be extremely inefficient. A lot of research has gone into finding better merge algorithms in terms of time and IO-complexity: *MPMGJN* (Zhang et al. 2001), *EE/EA-join* (Li & Moon 2001), *tree-merge* and *stack-tree* (Al-Khalifa et al. 2002), *Anc Des* (Chien et al. 2002), *PathStack* and *TreeStack* (Bruno et al. 2002), *XR-tree* and *XR-stack* (Wang & Ooi 2003), *TS-Generic* (Jiang et al. 2003), and *TJFast* (Lu et al. 2005). Even though these algorithms are very efficient in terms of their input and output, many systems which use them perform a lot of unnecessary work. Assume for example the query `/a/b/c/"foo"`, and that `a` is the first element of the path for half

¹Let `path/"term"` be short for `path[normalize-space(text())="term"]`.

of the data values stored in the database, while *c* is seen only a few times. To do the merge, either the entire occurrence list for *a* must be read from disk, or every occurrence of *c* must be looked up in some data structure for *a* on disk. It is obvious that methods which utilise the varying selectivity of the query elements or the full paths will be faster.

Various methods which use index structures other than inverted lists have been proposed. The *index fabric* (Cooper et al. 2001) maintains a layered *Patricia trie* of all paths seen in the data. It is organised in multiple levels, so that a path query will result in a number of lookups (block) logarithmic in the size of the total data. A problem with this index structure is that only queries starting at the document root element are efficiently supported.

When the structure of XML documents is highly regular, the number of unique paths seen in a collection will be very small compared to the total size of the data. This set of unique paths can often fit in main memory, and be searched very efficiently. The first use of *path indexing* known to the author is by Bertino & Kim (1989), while perhaps the best known is the *DataGuide* (Goldman & Widom 1997) used in the Lore DBMS for semistructured data.

Let a *path summary* denote a summary of all paths which is not indexed for fast matching. Perhaps the simplest use of path summaries is by Buneman et al. (2005), where all paths are extracted from the data, and maintained as an in-memory “skeleton”. For each path seen, there is an on-disk vector containing all terms and values seen below instances of this path. Weaknesses of this approach is that a full search for matching paths in the skeleton is required when paths do not start with the root element, and that a brute force (or binary) search through the vector is necessary for queries with value predicates. The *ToXin* system (Rizzolo & Mendelzon 2001) improves the latter by maintaining for each path an index over the values seen below it. The strength of *ToXin* is an efficient matching of twig queries, by storing navigational information for the data in the nodes in the index. A further improvement is *ToXop* (Barta et al. 2005), in which query plans are made based on the selectivity on the path query elements, and clever combinations of merges and searches are used. A potential weakness is that if a query does not start with a root element, a brute force search through the path summary is required to match the path expression.

An enhancement over a brute force search through the summary is to make an inverted index over the paths on their tags. Given a path query, look up the individual tags in a path index and merge the results. This is used in *SphinxX* (Poola & Haritsa 2007), where there is a value index for each path (as in *ToXin*). In the case where the path index is of considerable size, the merging can be costly. The systems *APEX* (Chung et al. 2002) and *XIST* (Runapongsa et al. 2004) address this by maintaining index entries for sub-paths of lengths greater than one on demand.

A simple and elegant system for XML indexing using path indexing in a RDBMS is *XRel* (Yoshikawa et al. 2001). One of the four tables used is a mapping from paths to integer identifiers. All text and values indexed have a reference to the path under which they reside, and path matching is done using simple LIKE queries with wild-cards in the path table. Similar solutions is used in many systems based on RDBMS.

A problem with keeping a separate value index for each path is cases where many paths match the query. The worst case scenario is when the query consists of only a value predicate. This results in many disk accesses, unless the indexes are stored in some interleaved fashion, grouped on the value key. An alternative is to have a single value index, where

the occurrences of a value are stored with their parent path ID. After the entry for a value has been found, the occurrences are filtered on matching path IDs. If the occurrence list is large, it can be stored sorted on path ID, and pointers into the list can be used to avoid having to read all of it (known as *skip lists*).

When the path summary fits in main memory, the choice of index structures which are suitable for implementing it is greater than if it would have to reside on disk. One structure which is only efficient in main memory is the suffix tree (McCreight 1976). *PIGST* (Zuopeng et al. 2007) is a system maintaining a generalised suffix tree as the path index². See Section 2.4 for a description of this solution. A more common use of (often pruned) suffix trees is selectivity estimation for optimising query plans (Aboulnaga et al. 2001, Chen et al. 2001).

A method very different from *node* and *path indexing* is *sequence indexing*, where all documents are converted to a sequence representation, and searching is done by subsequence matching. *ViST*³ (Wang et al. 2003) is a system using this approach. An advantage is that searching for twig queries can be done without merging partial results. A problem with *ViST* is that the index has quadratic size in the worst case, if the trees indexed are very deep. *PRIX* (Rao & Moon 2004) solves this by taking a different approach to the sequencing, using Prüfer sequences. Wang & Meng (2005) use a representation similar to in *ViST*, but using a more clever sequencing they optimise for smaller indexes and faster queries. The querying process also becomes much simpler than in *ViST* and *PRIX*.

1.2 Contributions

This article describes how to do efficient path matching. It is assumed that there is an overlying system similar to what is common when using *path indexing* (see Section 2.1).

- It is shown that to combine an inverted index for the path summary with brute force search is in practise much faster than merging path element hits. The methods introduced exploit the varying selectivity of the query path elements.
- It is shown how the use of a generalised suffix tree can be enhanced by adding statistics to the tree nodes, and changing the way searches are performed. Multiple entry points into the query are partially evaluated in parallel, depending on the evaluation cost.
- Many path index implementations are compared, using paths generated from statistical models and from DTDs.

2 Path index implementation

Below follows descriptions of various solutions for implementing path summaries.

2.1 Assumptions

This article only addresses the implementation of the path index, and assumes that an overlying system with the following design is using it: All values and terms seen in the document collection are indexed by

²The authors make some extensions which makes the suffix trees super-linear in size, seemingly without considering this.

³Stands for Virtual Suffix Tree, but only due to a misconception from the author.

<a>	1
foo	1.1
foo	1.2(.1)
	1.3
<c>foo</c>	1.3.1(.1)
	1.3.2
bar	1.3.2.1
<a>	1.3.2.2
bar	1.3.2.2.1
	1.3.2.2.2
	
	
	
foo	1.4(.1)
<c>bar</c>	1.5(.1)
	

Figure 1: Example XML document. Dewey order encoding of elements shown on the right.

- | | |
|----|------------|
| 1. | /a |
| 2. | /a/b |
| 3. | /a/b/c |
| 4. | /a/b/b |
| 5. | /a/b/b/a |
| 6. | /a/b/b/a/b |
| 7. | /a/c |

Figure 2: Enumeration of unique paths seen in XML in Figure 1. This is the set of paths which would be indexed by a path index.

ordinary inverted lists. Stored in each entry is information encoding document ID, position in the document, the values parent path identifier, and a local specifier for the path instance (range based or prefix based). Figure 3 shows the value index for the example XML document in Figure 1, with path ID and Dewey order encoding of path instance shown. Document ID and position is omitted for brevity. The enumeration of the paths is shown in Figure 2.

Given a non-branching XPath query with a value predicate, all paths matching are found with the path index. The value is then looked up in the value index, and the hits are filtered with the set of matching paths. For the query `/a//c/"foo"`, the paths matching `/a//c` in the example XML are number 3 and 7. The only occurrence in the lists for "foo" with a path which matched is (3 1.3.1.1). In the case of XPath queries without value predicates, an index for occurrences of XML tags should be maintained in addition. The value index may also have the occurrence lists split/sorted on path ID for faster filtering, as in *ToXin* (Rizzolo & Mendelzon 2001) and *Sphinx* (Poola & Haritsa 2007).

It is assumed that representatives for the unique paths seen in a document collection fit in main memory, and further any index structure which is linear in their size. Only "simple" path expressions are considered, not twigs. An example of a twig XPath query is `/a/b[c/"foo" and b/"bar"]`. It is assumed that a system using the path index here would perform two queries (one for each branch in the twig), and merge the results. Here the merge would check for a common prefix of length three in the Dewey encoding of the paths. Note that the problem is much more involved in general. Unordered tree inclusion in general is even NP complete (Kilpeläinen 1992). The reason twigs are not treated here, is that in most cases, the leaves of the twigs will be value predicates (as in the given query), which will have to be looked up in the value index in any case, given the overall system design.

Below follows the descriptions of various path index data structures and matching approaches.

"foo":	1 1.1
	2 1.2.1
	3 1.3.1.1
	2 1.4.1
"bar":	4 1.3.2.1
	5 1.3.2.2.1
	7 1.5.1

Figure 3: Value index for example XML from Figure 1. Storing path ID and path instance Dewey order. Document ID and position omitted.

a:	1,1 2,1 3,1 4,1 5,1 5,4 6,1 6,4 7,1
b:	2,2 3,2 4,2 4,3 5,2 5,3 6,2 6,3 6,5
c:	3,3 7,2

Figure 4: Inverted lists for paths seen in example XML. Storing path ID and position within path.

2.2 Brute force search

The simplest way to implement a path index is to store the paths seen in a list, and perform brute force searches for path expressions through this list. Given regular path expressions, a *deterministic finite automata* (Aho et al. 1986) (DFA) for the query can be built. A DFA can be exponential in the size of the query, but for most cases queries can be considered to be of constant length. Using a DFA gives a linear time scan through the data. For document collections with large data, but small schema, a brute force search may be a sufficient solution, as the scan through memory is relatively cheap compared to the disk accesses needed for the lookup into the value index.

2.3 Inverted list solutions

When inverting paths, each tag in a path is treated as a symbol. The index will contain for each symbol a list of positions in which it occurs, given as path ID and position within path. An index for the paths in Figure 2 is shown in Figure 4. Whether the index should store pairs of path ID and position, or store the path ID and a list occurrence positions within the path, depends on the expected lengths of these lists. In an implementation not using compression, an additional integer would be needed for storing the length of each list. This means the latter approach is more space efficient when the expected list length is greater than two. This could happen with recursive document schemas. The approach using simple pairs was chosen for simplicity in the implementations used here.

Given a path query using only the child axis, each element is looked up, and the results are merged where the elements are adjacent in paths. Assuming the query `//a/b/c`, merging left to right, first merge hits for `a` and `b`, and keep all hits with adjacent elements. The reason all hits are needed, even though the final output is only path IDs, is that which hits for `a/b` have adjacent hits for `c` is not known. After merging with `c` in the example, a match in path 3 is left, from position 1 to 3.

For the descendant axis, hits need not be adjacent, only in the correct order. Given that the element hits are merged left to right, only the hit with the leftmost right border need to be passed on to the next step in the merge. For the query `//a//b//c`, first merge hits for `a` and `b`, and keep at most a single match in each path, one with `b` as far to the left as possible. Then merge this hit set with the hits for `c`.

For queries using both the child and descendant axis, the hits for elements in a parent-child rela-

tionship should be merged first, then elements in an ancestor–descendant relationship. This is because the former needs all intermediate hits, while the latter does not.

2.3.1 Indexing tuples

The performance for path queries using the child axis can be greatly improved by indexing pairs, triples, or even longer substrings in the inverted lists for the paths. What is indexed can be decided dynamically, as in *APEX* (Chung et al. 2002) or *XiST* (Runapongsa et al. 2004), or statically, as is done here for simplicity. If the size of the data (in this case the paths) is large compared to the alphabet, the space overhead associated with starting a list in the index is small compared to the size of the list contents. In this case, an index for all pairs will not require much more space than an index for all single elements.

It is expected that using pairs or triples will greatly reduce query cost in practise, as these probably will have much better selectivity than single elements.

2.3.2 Extending possible hits

Given queries using the child axis, a simple trick can be used to improve the performance. Assume only single elements are indexed, and the query is `/a/b/c`. If `a` is the root element of every second path, `b` exists in around half the paths, but `c` is seldom seen, the size of the result will be very small compared to the total cost of the merges. The cost of merging large and small hit sets can be reduced by performing binary searches in the large set. The merges can also be done out of order to reduce the cost.

A simpler and more efficient solution is possible. Since all paths reside in main memory, checking single elements in the paths is very cheap. Take the hits for the most selective element, and for each one, check whether it is preceded and succeeded by the needed elements. This avoids merging with larger sets due to poorer selectivity. The method can be combined with indexing pairs and triples.

2.3.3 Estimate, choose, brute

Expensive merges on the descendant axis can also be avoided when a part of the query has good selectivity. Assume the XPath query `/a//c`, where `c` has good selectivity, but `a` does not. As the paths are relatively short strings, a brute force search through the set of paths containing `c` should be more efficient than a merge. The memory management overhead of handling intermediate hit sets is also avoided.

2.4 Suffix tree solutions

A *generalised suffix tree* for a set of strings is a compacted trie for all suffixes of the strings (McCreight 1976). An example tree is shown in Figure 5. This structure can be built in time and space linear to the total length of the strings for constant and integer alphabets (Farach 1997). For general alphabets the complexity is $O(n \log |\Sigma|)$. The implementation used in this article combines the child arrays from Grimsmo (2005, 2007) and hashing. The index can decide whether a given string exists as a substring in the set in expected time linear to the length of the string, and all hits can then be extracted in time linear to their number. The set of paths in a path summary can be seen as a set of strings, where XML tags are string symbols, and indexed with the suffix tree. This requires that the suffix tree implementation can handle the possibly large alphabet of XML tags.

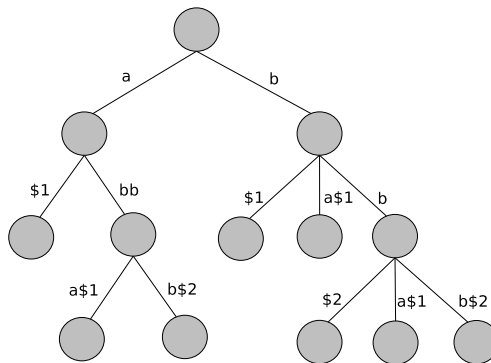


Figure 5: Generalised suffix tree for the strings `abba` and `abbb`.

If this is not the case, the paths can be spelled out separated by delimiters, and these longer strings can be indexed. In the implementation used here, tags were mapped to an integer alphabet used as string symbols.

If only the child axis is used all matches are contiguous sub-paths, and the node whose subtree represents all hits can be found in optimal time. But not all occurrences of the sub-path are needed, only the set of paths containing them. As an example, `a/b` occurs twice in the path `/a/b/b/a/b`, but given the query `//a/b/"foo"`, only the fact that it occurs is of interest. In *PIGST* (Zuopeng et al. 2007) this is solved by storing in each internal node of the tree the set of path IDs seen in the subtree below. For random paths from a uniform distribution, the average ratio between the size of the subtree and the size of the list of path IDs will be inversely proportional to the average path length. Another argument for their solution, is that the nodes in a suffix tree built with any known linear construction algorithm have bad spatial locality, while the path IDs stored in the lists in *PIGST* have perfect locality. This is of importance also in main memory, because of the cache effects of modern computers. No bounds on space usage or construction time for their extended suffix tree is given (Zuopeng et al. 2007), but both are super-linear, even in the average case (Grimsmo & Bjørklund 2007). Finding the set of strings which contain a given substring is known as the *document listing problem*, and can actually be solved optimally with linear preprocessing (Muthukrishnan 2002).

2.4.1 Intersect, brute

The straight forward way to search for a path expression using the descendant axis, is to first do a search for the node representing the first part of the query (using only the child axis), and then do a full recursive search of the subtree below. To avoid this, *PIGST* does a separate search for each part of the query, intersects the resulting sets of path IDs, and performs a brute force search through the set of possible paths. Note that this merging on the descendant axis is different from what is described earlier in Section 2.3, as sets of path IDs are intersected, not sets of hits in paths, where in-path order is of importance.

A variant of this is to take only the smallest set of path IDs, and perform a brute force search through the respective paths. This is similar to what was described for inverted files in Section 2.3.3. One difference is that if a part of a query using only the child axis has been matched in the tree, the size of the hit set does not need to be estimated, as it is known. Another is that the set of matches is extracted without overhead, no matter how long the query part is. For

inverted lists, merging or hit expansion was necessary if the part was longer than the tuples indexed.

Skipping the intersection and just doing the brute force search through the smallest set should pay off when the total length of the paths in the smallest set of possible paths is less than the number of path IDs in the largest set. This could happen often if the paths were generated by a source with skewed distribution.

2.4.2 Selective suffix tree traversal

Below follows the description of a novel algorithm using multiple entry points into the query. Pseudo-code is given in Algorithm 1. The nodes of an ordinary generalised suffix tree are each extended with a number giving the size of the subtree below the node. This allows for more intelligent traversal of trees and queries. Two variants are used, where the second uses two suffix trees.

```

Input: path expression  $P$ , suffix tree  $ST$ 
Output: set of matching paths
 $Q \leftarrow \text{PriQueue}(\text{getEntryPoints}(P));$ 
while not complete(front( $Q$ )) do
   $ep \leftarrow \text{pop}(Q);$ 
   $next \leftarrow \{\};$ 
  foreach  $p \in ep.positions$  do
     $next \leftarrow next \cup \text{advance}(p);$ 
  end
   $c \leftarrow 0;$ 
  foreach  $p \in next$  do
     $c \leftarrow c + \text{nextAdvanceCost}(p);$ 
  end
   $ep.positions \leftarrow next;$ 
   $ep.advanceCost \leftarrow c;$ 
   $\text{push}(Q, ep)$ 
end
 $ep \leftarrow \text{front}(Q);$ 
return  $ep.matches;$ 

```

Algorithm 1: Selective suffix tree traversal

For the first variant the number of entry points into a query is equal to the number of parts separated by the descendant axis. Given the query `/a/b//c//d/e`, there are entries starting at `a`, `c` and `d`.

All entry points are kept in a priority queue. They are ordered on the expected cost of evaluating them one step further. As soon as one is completely evaluated, the matches are extracted. Each entry point may during evaluation be at multiple positions in the suffix tree, if wild-cards or the descendant axis have been used. The cost of moving an entry point forward in the query is the sum of the costs for moving downward at each position held in the tree, with a reduction for having advanced further into the query. Evaluating a step of the child axis costs 1, except when the next symbol is a wild-card, where the cost is equal to the number of children of the current node in the suffix tree. For the descendant axis, the cost of moving one step forward in the query is equal to the size of the subtree below the current node. For an entry point which started inside the query, and is expanded all the way to the end, the cost of evaluating it “one step further” is an estimate of the cost of a brute force search through the paths with the partial match, which must be done to get a full match.

The other variant of this method also uses a suffix tree for the reverse representation of the paths. It has additional entry points moving backwards from the last element in each part of the query, doing the matching in the second suffix tree. For the example

query, there would also be entry points moving backwards from `b`, `c` and `e`. A variant using only the suffix tree for the reversed paths is also included in the tests.

2.4.3 Skipping leading wild-cards

A simple variant of the basic use of a suffix tree can be used when the query starts with a wild-card. The leading wild-cards can be omitted from the query, and after the hits have been retrieved, they can be filtered on their starting positions in the paths.

3 Experimental results

The tests were run on an AMD64 3500+ running Linux 2.6.16 compiled for AMD64. All tested implementations were written in Java, and run with 64-bit Sun Java 1.5.0_06. As the Java virtual machine often shows a radical speedup from “warming up” (optimising byte-code), and as many of the solutions shared code, some measures had to be taken to give a fair treatment. Each test was run repeatedly with all implementations, until neither of them showed a deviance of more than 2% in total running time for the test from the last attempt. This ensured that all implementations got a proper and fair warmup.

Some care also had to be taken when measuring the memory usage, as Java relies on garbage collection. The garbage collector was called multiple times before the indexing process started, and the base memory usage was measured⁴. It was called again multiple times after the indexing finished, and the difference to the base memory usage was then recorded. If the garbage collector was run only a single time, the space usage measured differed greatly between runs.

3.1 Data generation

The paths used in the tests were generated both from statistical models and DTDs. A uniform distribution and Zipf distributions were used, in addition to first order Markov models with underlying Zipf distributions.

The DTDs used for path generation were taken from the benchmarks DBLP (Ley 2007), GedML (Kay 1999), Michigan (Runapongsa et al. 2003), Xbench (Yao et al. 2003), XMach-1 (Böhme & Rahm 2001), XMark (Schmidt et al. 2001) and XOO7 (Bresnan et al. 2003). Paths were generated by a breadth first search through the space of possible paths defined by the DTDs. Some of the DTDs were not used alone, as they were small and/or non-recursive. In tests using multiple DTDs, a breadth first search through their complete space was done, such that all paths of length k were generated before any path of length $k + 1$. For the data from statistical distributions, path lengths were drawn randomly from 1 to 10.

Queries for paths from DTDs were generated as follows: The length of the query was drawn from its distribution (Default: uniform from 1 to 5). Then, for each position in the query, the use of the child or descendant axis was chosen. (Default: $p(\text{desc}) = 0.3$). If the descendant axis was not used at all, a random path of the specified length was drawn, and used as query. If not, a random path of *at least* the specified length was drawn. Then the child axis operators were inserted into the query at random locations, and random path elements next to them were removed until the query had the specified length. This procedure ensured that all queries related to the generated

⁴`Runtime.totalMemory() - Runtime.freeMemory()`

data. The probability that a query required matches to start at a root element was set to 0.5. All queries were required to match the end of a path. Finally, the probability that a path element was substituted with a wild-card was by default 0.1.

3.2 Methods tested

The following methods were used in the tests :

Br Brute force search through all strings. (See Section 2.2)

MgInv Inverted lists and merging. (2.3)

MgIe1 Inverted lists, selective entry point in contiguous part, expanding on child axis, merging on descendant axis. (2.3.2)

MgIe2 Indexing singles and pairs. (2.3.1+2.3.2)

MgIe3 Indexing singles, pairs and triples. (2.3.1+2.3.2)

EsIe[1,2,3] Estimating contiguous part with fewest hits, extracting possible paths, filtering with brute force. (2.3.2+2.3.3)

St Straight forward use of suffix tree. (2.4)

Ss Suffix tree, skipping leading wild-cards, and filtering on start position in string. (2.4.3)

InSe Suffix tree with path ID lists in internal nodes. Intersection of path ID sets on descendant axis and brute force through result. As described in (Zuopeng et al. 2007). (2.4.1)

EsSe Suffix tree with path ID lists in internal nodes. Finding the contiguous part of the query (no descendant axis) with fewest matches, and brute force search through the corresponding set of paths. (2.3.3+2.4.1)

Sm[f,r,2] Suffix tree(s) with multiple entry points. Testing single tree with *forward* strings, *reversed* strings, and *two* trees, one forward and one reversed. (2.4.2).

Smfe Suffix tree enhanced with path ID lists, using multiple entry points, using only forward tree. (2.4.1+2.4.2)

3.3 Tests using various data sources

Table 1 shows query performance for the tested methods. 10000 paths were indexed, drawn from the different data sources. 5000 queries of length 1 to 5 were run, with a 0.3 probability of using the child axis and 0.1 probability of wild-cards. See later tests for variations over this. Table 2 shows more measures for the test using all the DTDs.

The brute force solution (*Br*) serves as a base case for comparison. It is faster than some methods on many of the tests, as these methods have to merge very large sets. The performance is also related to the query selectivity. For the test with poorest average selectivity, the fastest method is only five times faster than the brute force search, while where the selectivity is best, the fastest method is more than 50 times faster. The simplest merging method is *MgInv*, which looks up every single (non-wild-card) element of the path query, and merges the results, first on the child axis, then on the descendant axis. In the case of uniform data ($|\Sigma| = 100$) it is comparable with the brute force solution, but it is much slower on many other tests.

The methods named *MgIe** improve this by finding entry points into the contiguous parts of the queries, keeping the hits that expand with a match, and then merging only on the ascendant axis. These methods are only faster than *Br* on the artificial tests

with rather uniform data. As the probability of using the descendant axis is 0.3, there will frequently be parts of the query still with low selectivity after expanding over the child axis. Notice that also indexing pairs (*MgIe2*) gives a significant speedup, while the speedup from indexing triples (*MgIe3*) is less dramatic. Table 2 shows that *MgIe2* uses almost twice as much memory as *MgIe1*, and *MgIe3* almost three times as much, as expected. The total memory used for indexing 10000 paths with *MgIe3* was measured to 2.6 MB. The methods which combine inversion and brute force (*EsIe**) have a greater speedup from indexing pairs and triples. They also have a significant speedup over merging in general. On the test using multiple DTDs, *EsIe3* is more than eight times faster than *MgIe3*. Indexing triples does not help the latter much if the shortest contiguous parts of the query are single elements with poor selectivity.

The straight forward use of a suffix tree (*St*) has better performance than any of the merging variants (*Mg**), but is slower than the combinations of indexing, selectivity estimation and brute force (*Es**). When the suffix tree encounters use of the descendant axis, it must traverse an entire subtree, which is a costly operation if the first part of the query was not very selective. The space usage for the suffix tree is similar to that of indexing triples (**Ie3*). The improvement of *Ss* over *St* on some of the tests comes from queries starting with wild-cards. *St* must branch to every child of the root node in the tree, while *Ss* skips this part of the query, and filters hits on their starting positions in the paths afterwards. The reason *Ss* is sometimes slower is probably the overhead of the filtering. *St* can be fast when a query starts with a wild-card if the next elements are very selective, and the branching effectively cut off.

The method *InSe*, based on *PIGST* (Zuopeng et al. 2007), is faster than *St* on all, and *Ss* on most of the tests. It uses a suffix tree enhanced with path ID lists, path set intersection and brute force search, as described in Section 2.4.1. As predicted, the related method *EsSe* is considerably faster on the tests with non-uniform data. It skips the intersection, and performs the brute force search through the smallest set, exploiting the varying selectivity of the parts of the query. It should be noted that *EsIe3* is faster than *InSe* on all the tests, while *EsSe* has a similar performance. *EsIe3* also uses less space and has faster index construction (See Table 2).

The suffix trees using multiple entry points into the query (*Sm[f,r,2]*) have very good performance. *Smf* is more than three times faster than *InSe* on the test using multiple DTDs, and also faster than *EsIe3*. Using the forward representation of the paths (*Smf*) is more efficient than using the backward representation (*Smr*). There are two reasons for this. The first is the probabilities for requiring match in the beginning and end of a path, which are 0.5 and 1.0. When these were swapped *Smr* was equivalently faster on tests using uniform and Zipf data. For paths generated from DTDs and Markov chains, *Smf* was still faster. Notice that *Smr* uses more memory. Not all of it comes from storing the reverse strings. The number of internal nodes in a suffix tree is upper bounded by the size of the input, but depends on its characteristics. A larger number of internal nodes gives more expensive tree traversals. It seems the reverse paths from DTDs have Markov properties that give a larger tree.

Building two suffix trees and using both forward and reverse entry points (*Sm2*) does not increase performance. More entry points are partially evaluated, seemingly without reducing the total cost. It is possible that a more intelligent and well tuned implementation would give better results. *Sm2* used the most

	%	Br	MgInv	MgIe1	MgIe2	MgIe3	EsIe1	EsIe2	EsIe3	St	Ss	InSe	EsSe	Smf	Smr	Sm2	Smfe
U ₁₀	1.8	1068	5637	1426	755	715	807	206	164	350	365	241	150	148	224	148	99
U ₁₀₀	0.2	1012	937	164	82	81	96	25	24	81	51	68	57	22	29	26	19
Z _{100,0.7}	0.4	1021	1719	306	186	182	117	41	37	126	106	92	55	35	44	41	28
Z _{100,1.0}	0.9	1038	3658	691	448	437	223	89	77	249	241	156	76	73	90	77	55
Z _{100,1.3}	2.3	1094	8067	1541	1097	1052	483	219	181	543	549	320	157	165	275	166	117
MZ _{100,0.7}	0.2	1013	957	177	94	94	96	28	26	92	73	64	48	23	33	26	19
MZ _{100,1.0}	0.3	1020	1132	230	139	136	108	35	33	144	128	76	46	27	38	30	21
MZ _{100,1.3}	0.5	1038	1394	307	205	204	137	48	46	215	205	92	50	36	53	40	25
DBLP.dtd	2.4	1241	9455	2283	1724	1673	804	326	271	985	1012	559	262	239	533	286	170
GedML.dtd	1.2	1181	7504	1583	1205	1205	460	119	117	731	737	394	92	75	151	98	56
XMark.dtd	3.2	1347	9754	2382	1731	1689	942	401	359	1097	1137	590	339	336	1074	307	263
*.dtd	0.6	1097	3222	715	600	599	161	78	73	365	369	201	62	61	115	85	50

Table 1: Microseconds per query, average. Testing uniform distribution (parameter $|\Sigma|$), Zipf distribution ($|\Sigma|, s$), a first order Markov model with underlying Zipf ($|\Sigma|, s$), and various DTDs. Second column shows average query selectivity per cent.

	Br	MgInv	MgIe1	MgIe2	MgIe3	EsIe1	EsIe2	EsIe3	St	Ss	InSe	EsSe	Smf	Smr	Sm2	Smfe
$\mu s/q$ dev	336	3214	1107	1063	1074	304	220	231	680	691	355	181	170	365	245	132
$\mu s/path$	0	2	2	3	5	2	3	6	14	14	21	21	14	17	33	22
b/elem	4	18	18	33	50	18	33	50	48	48	76	76	48	69	114	76

Table 2: Indexing paths from all DTDs. Showing microseconds per query standard deviation, microseconds per path when indexing, and bytes per path element in the complete index.

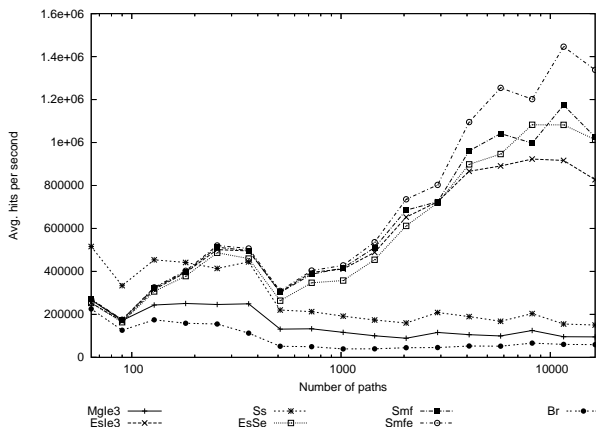


Figure 6: Increasing number of paths. Hits per second.

memory of all implementations, totalling to 5.8 MB on the test will all DTDs.

Smfe combines features from *Smf* and *InSe*, and turns out to beat all methods on query performance. A drawback from *Smf* is the increased construction time and memory usage, which comes from using the data structures from *InSe*.

In the subsequent tests, only the fastest representatives from each group of implementations are shown.

3.4 Increasing number of paths

Figure 6 shows how the number of hits returned per second changes as the number of paths indexed increases. The paths were generated from all the DTDs, and otherwise the default parameters were used. Hits are counted instead of queries to get a better perspective of what happens when the size of the data is large. A “higher is better” measure is used to improve the visualisation of the difference between the best methods.

Smfe, *Smf*, *EsSe* and *EsIe3* out-scale the other methods by a good margin, with the first showing the best performance. The reason for the various drops and rises in the graph may be that the distribution of the paths generated from the DTDs change as the maximal path length increases. A similar test on paths from a Zipf distribution did not show the

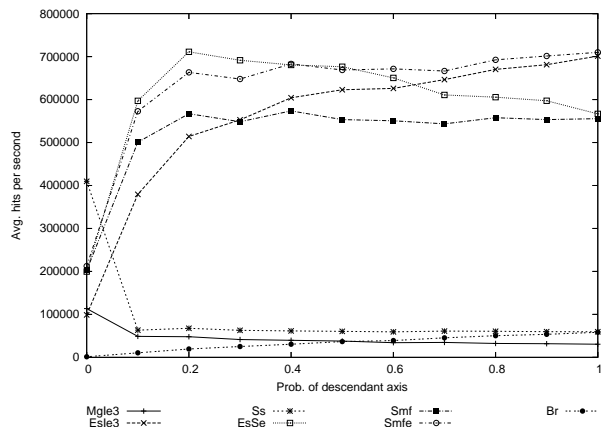


Figure 7: Increasing probability of descendant axis. Hits per second.

same drops.

3.5 Descendant axis

Figure 7 shows hits returned per second as the probability of using the descendant axis increases. The paths were generated from the DTDs, and the default parameters were used, except that the probability of wild-cards was set to zero. This was to isolate the effect of using the descendant axis. 10000 paths were indexed. Here *EsSe* is fastest when the probability of using the descendant axis is low, while *Smfe* is fastest when it is high. *EsSe* depends on finding contiguous parts of the query with good selectivity, which is harder when all parts are very short.

Note that the simple use of a suffix tree (*Ss*) has the best performance when there is no use of the descendant axis, but poor performance otherwise. The other methods using suffix trees could switch to the simple search technique when this is expected to be profitable.

3.6 Wild-cards

Increasing the probability of wild-cards gives different results than increasing the use of the descendant axis, as shown in Figure 8. The descendant axis was not used at all in this test. Here the suffix trees are much faster than the other methods. For the trees,

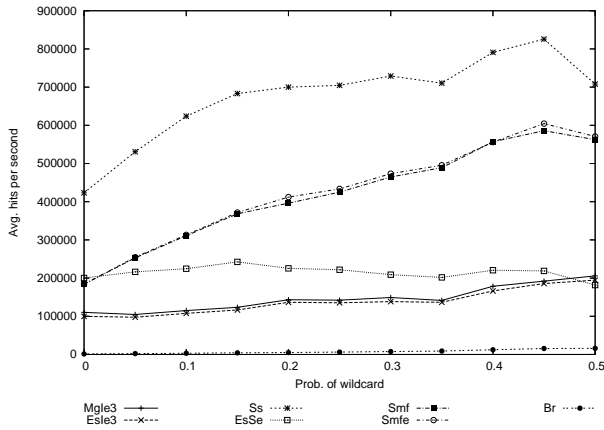


Figure 8: Increasing probability of wild-card. Hits per second.

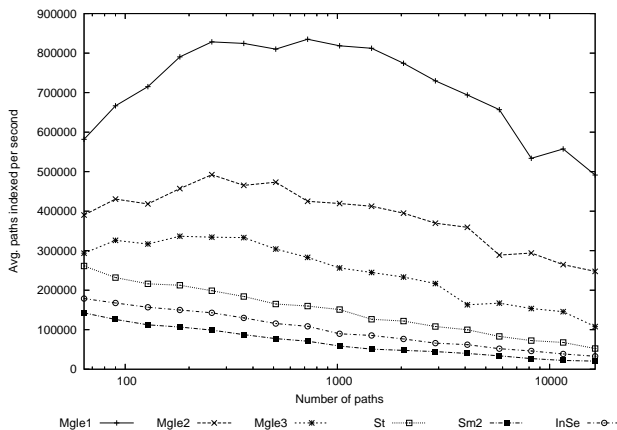


Figure 9: Paths indexed per second.

branching on a wild-card is much less critical than branching on the descendant axis, as the former is cut off as soon as a mismatch is seen, while the latter introduces a full search of the subtree.

It is interesting to note that the simple use of a suffix tree, only skipping leading wild-cards (*Ss*), here is much more efficient than the implementation supporting multiple entry points into the query (*Smf*), even though only a single entry point is created. The realisation of the search automata in *Ss* is just a recursive program, while *Smf* maintains a set of state objects, giving a lot of overhead.

3.7 Index construction

Figure 9 shows the indexing performance of the various indexes as the number of paths increases. Single representatives for methods using the same data structure were tested. *Mgle1* represents *MgInv* and *Esle1*, *Mgle2* represents *Esle2*, *Mgle3* represents *Esle3*, *St* represents *Ss* and *Smf*, and finally *InSe* represents *EsSe* and *Smfe*. The methods tested are all asymptotically linear in the worst case except *InSe*. The performance drop observed is probably due to the overhead of memory management and cache effects. The construction of the suffix trees is slower than construction of inverted lists, but as the time cost of adding a path is less than 30 μ s, this would constitute a negligible part of the cost of indexing XML in a complete system, if it is assumed that the path index can reside in main memory, while the value index must reside on disk.

4 Conclusion and future work

The most advanced method using suffix trees (*Smfe*) is the winner on query performance in these tests. Whether this should be chosen as the path index component in a larger system, depends on the amount of main memory available and on whether or not the performance of the path index significantly impacts the performance of the complete system. Another issue is the complexity of the implementation. The suffix tree itself is a rather complex structure, and its use as described here may be hard to grasp.

The method combining inverted lists, extension over the child axis and brute force (*Esle3*), would be the authors' choice when implementing a larger system. It is conceptually simple, easy to implement, and has good performance. If memory usage is an issue, *Esle2* or *Esle1* could be used. These methods could also be modified to work on disk when the path index does not fit in main memory. The paths themselves could be stored in the entries in their inverted lists, allowing the extension technique to work, at the cost of using more disk space and IO.

In future work, the authors would like to add the fast path summaries introduced to an existing system for indexing XML, and compare this with various implementations, both with and without path summaries, such as *Lore* (Goldman & Widom 1997), *ToXin* (Rizzolo & Mendelzon 2001) *XISS* (Li & Moon 2001), *XIST* (Runapongsa et al. 2004), *Ctree* (Zou et al. 2004), *SphinX* (Poola & Haritsa 2007) and *MXI* (Yan & Liang 2005).

Acknowledgements

The author would like to thank Øystein Torbjørnsen, Svein-Olaf Hvasshovd and Truls Amundsen Bjørklund for helpful feedback on this article.

References

- Aboulnaga, A., Alameldeen, A. R. & Naughton, J. F. (2001), Estimating the selectivity of XML path expressions for internet scale applications, in 'Proc. VLDB', pp. 591–600.
- Aho, A., Sethi, R. & Ullman, J. (1986), *Compilers, Principles, Techniques and Tools.*, Addison Wesley, Reading.
- Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J. & Srivastava, D. (2002), Structural joins: A primitive for efficient XML query pattern matching, in 'Proc. ICDE', pp. 141–152.
- Barta, A., Consens, M. P. & Mendelzon, A. O. (2005), Benefits of path summaries in an XML query optimizer supporting multiple access methods, in 'Proc. VLDB', pp. 133–144.
- Bertino, E. & Kim, W. (1989), 'Indexing techniques for queries on nested objects', *IEEE Transactions on Knowledge and Data Engineering* 1(2), 196–214.
- Böhme, T. & Rahm, E. (2001), 'XMach-1: A benchmark for XML data management', *Proc. German database conference BTW* pp. 264–273.
- Bressan, S., Lee, M., Li, Y., Lacroix, Z. & Nambiar, U. (2003), 'The XOO7 benchmark', *Proc. EEXTT'02* pp. 146–147.
- Bruno, N., Koudas, N. & Srivastava, D. (2002), Holistic twig joins: Optimal XML pattern matching, in 'Proc. SIGMOD', pp. 310–321.

- Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R. & Viglas, S. D. (2005), Vectorizing and querying large XML repositories, *in* 'Proc. ICDE', pp. 261–272.
- Chen, Z., Jagadish, H., Korn, F., Koudas, N., Muthukrishnan, S., Ng, R. & Srivastava, D. (2001), 'Counting twig matches in a tree', *Proc. ICDE*.
- Chien, S., Vagena, Z., Zhang, D., Tsotras, V. & Zaniolo, C. (2002), 'Efficient structural joins on indexed XML documents', *Proc. VLDB* **2**, 263–274.
- Chung, C.-W., Min, J.-K. & Shim, K. (2002), APEX: An adaptive path index for XML data, *in* 'Proc. SIGMOD', pp. 121–132.
- Cooper, B., Sample, N., Franklin, M. J., Hjaltason, G. R. & Shadmon, M. (2001), A fast index for semistructured data, *in* 'Proc. VLDB', pp. 341–350.
- Farach, M. (1997), Optimal suffix tree construction with large alphabets, *in* 'Proc. FOCS', pp. 137–143.
- Goldman, R. & Widom, J. (1997), DataGuides: Enabling query formulation and optimization in semistructured databases, *in* 'Proc. VLDB', pp. 436–445.
- Gottlob, G., Koch, C. & Pichler, R. (2005), 'Efficient algorithms for processing xpath queries', *ACM Trans. Database Syst.* **30**(2), 444–491.
- Grimsmo, N. (2005), Dynamic indexes vs. static hierarchies for substring search, Master's thesis, Norwegian University of Science and Technology.
- Grimsmo, N. (2007), On performance and cache effects in substring indexes, Technical Report IDI-TR-2007-04, NTNU, Trondheim, Norway.
- Grimsmo, N. & Bjørklund, T. A. (2007), On the size of generalised suffix trees extended with string ID lists, Technical Report IDI-TR-2007-01, NTNU, Trondheim, Norway.
- Grust, T. (2002), Accelerating XPath location steps, *in* 'Proc. SIGMOD', pp. 109–120.
- Jiang, H., Wang, W., Lu, H. & Yu, J. (2003), Holistic twig joins on indexed XML documents, *in* 'Proc. VLDB', pp. 273–284.
- Kay, M. H. (1999), 'GedML'. <http://users.breathe.com/mhkay/gedml/>.
- Kemper, A. & Moerkotte, G. (1992), 'Access support relations: an indexing method for object bases', *Inf. Syst.* **17**(2), 117–145.
- Kilpeläinen, P. (1992), Tree matching problems with applications to structured text databases, Technical Report A-1992-6, Department of Computer Science, University of Helsinki.
- Ley, M. (2007), 'DBLP XML Records'. <http://www.informatik.uni-trier.de/~ley/db/>.
- Li, Q. & Moon, B. (2001), Indexing and querying xml Data for regular path expressions, *in* 'Proc VLDB', pp. 361–370.
- Lu, J., Ling, T., Chan, C. & Chen, T. (2005), From region encoding to extended dewey: On efficient processing of XML twig pattern matching, *in* 'Proc. VLDB', pp. 193–204.
- McCreight, E. M. (1976), 'A space-economical suffix tree construction algorithm', *J. ACM* **23**(2), 262–272.
- Michiels, P., Mihaila, G. A. & Simeon, J. (2007), Put a tree pattern in your algebra, *in* 'Proc. ICDE', pp. 246–255.
- Muthukrishnan, S. (2002), Efficient algorithms for document retrieval problems, *in* 'Proc. SODA', pp. 657–666.
- Nestorov, S., Ullman, J. D., Wiener, J. L. & Chawathe, S. S. (1997), Representative objects: Concise representations of semistructured, hierarchical data, *in* 'Proc. ICDE', pp. 79–90.
- Poola, L. & Haritsa, J. (2007), 'Schema-conscious XML indexing', *Information Systems* **32**, 344–364.
- Rao, P. & Moon, B. (2004), Prix: Indexing and querying xml using prefix sequences, *in* 'ICDE '04: Proceedings of the 20th International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, p. 288.
- Rizzolo, F. & Mendelzon, A. (2001), Indexing XML data with ToXin, *in* 'Proc. WebDB', Vol. 2001.
- Runapongsa, K., Patel, J., Bordawekar, R. & Padmanabhan, S. (2004), XIST: An XML index selection tool, *in* 'Proc. XSym'.
- Runapongsa, K., Patel, J., Jagadish, H. & Al-Khalifa, S. (2003), The Michigan Benchmark: A microbenchmark for XML query processing systems, *in* 'Proc. EEXTT'02', Springer.
- Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J. & Busse, R. (2001), The XML Benchmark Project, Technical Report INS-R0103, CWI, Amsterdam, The Netherlands.
- Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E. & Zhang, C. (2002), Storing and querying ordered XML using a relational database system, *in* 'Proc. SIGMOD', pp. 204–215.
- Wang, H. & Meng, X. (2005), On the sequencing of tree structures for XML indexing, *in* 'Proc. ICDE', pp. 372–383.
- Wang, H. & Ooi, B. (2003), XR-tree: Indexing XML data for efficient structural joins, *in* 'Proc. ICDE', pp. 253–264.
- Wang, H., Park, S., Fan, W. & Yu, P. (2003), 'ViST: a dynamic index method for querying XML data by tree structures', *Proc. SIGMOD* pp. 110–121.
- Yan, L. & Liang, Z. (2005), 'Multiple schema based XML indexing', *Lecture Notes in Computer Science* **3619**, 891–900.
- Yao, B. B., Özsu, M. T. & Keenleyside, J. (2003), XBench - a family of benchmarks for XML DBMSs, *in* 'Proc. EEXTT'02', pp. 162–164.
- Yoshikawa, M., Amagasa, T., Shimura, T. & Uemura, S. (2001), 'XRel: a path-based approach to storage and retrieval of XML documents using relational databases', *ACM Transactions on Internet Technology* **1**(1), 110–141.
- Zhang, C., Naughton, J., DeWitt, D., Luo, Q. & Lohman, G. (2001), 'On supporting containment queries in relational database management systems', *SIGMOD Rec.* **30**(2), 425–436.
- Zou, Q., Liu, S. & Chu, W. W. (2004), Ctree: a compact tree for indexing XML data, *in* 'Proc. WIDM', pp. 39–46.
- Zuopeng, L., Kongfa, H., Ning, Y. & Yisheng, D. (2007), 'An efficient index structure for XML based on generalized suffix tree', *Information Systems* **32**, 283–294.