Computer Sciences Department

On the Integration of Structure Indexes and Inverted Lists

Raghav Kaushik
Rajasekar Krishnamurthy
Jeffrey Naughton
Raghu Ramakrishnan

Technical Report #1485

May 2003

UNIVERSITY OF
WISCONSIN
MADISON

On the Integration of Structure Indexes and Inverted Lists

Raghav Kaushik Rajasekar Krishnamurthy Jeffrey F Naughton Raghu Ramakrishnan

University of Wisconsin-Madison
{raghav,sekar,naughton,raghu}@cs.wisc.edu

Abstract

We consider the problem of how to combine structure indexes and inverted lists to answer queries over a native XML DBMS, where the queries specify both path and keyword constraints. We augment the inverted list entries to integrate them with a given structure index. We give novel algorithms for evaluating branching path expressions. Our experiments show the benefit of integrating the two forms of indexes.

We also consider the problem of incorporating relevance ranking into path expression queries. By integrating the above techniques with the Threshold Algorithm proposed by Fagin et al., we obtain instance optimal algorithms to push down top k computation.

1 Introduction

Recently, there has been a great deal of interest in the development of techniques to evaluate path expressions over collections of XML documents. In general, these path expressions contain both structural and keyword components. Unfortunately, while there are good techniques for evaluating structural components (for example, structural summaries), and good techniques for evaluating keyword components (for example, inverted lists), we are not aware of any techniques in the literature that work well on their combination. Furthermore, while an important application of this kind of query is to find the "top-k" documents that match the query, the published literature has not considered such "top-k" queries in conjunction with structural summaries. This paper attempts to fill this gap by proposing and evaluating an approach that merges structural summaries and inverted lists, and that works in the presence of top-k queries.

In more detail, consider the query //section/figure/title/"Graph". This query looks for the keyword "Graph" appearing at the end of a the sequence of structural containments //section/figure/title. To efficiently evaluate such queries, the XML query processing community first turned to inverted lists like those proposed by the structured document processing community [20]. Briefly, in the inverted list approach, the system builds inverted lists on the element names and text words appearing in the document. By performing "joins" of these inverted lists, one can verify containment relationships. To "follow" a path, the system computes a join for each step in the path. In our example, the system would compute three joins over the inverted lists for *section*, *title*, *figure*, and *"graph"*. This is essentially the approach taken in native XML query systems including Niagara [9] and Timber [8].

While inverted list processing has proven very effective for keyword searches in the information retrieval (IR) community, when applied to path expression queries over XML documents they are less universally effective. The problem is that evaluating a path may require many joins over large inverted lists, and these joins may be slow.

In view of this, there has been an orthogonal research effort into alternative indexes. Perhaps the most common and promising approach involves graph summarization [13, 16, 18]. Briefly, the idea here is to compute a "summary" of the XML documents. Ideally, this summary has the property that it is much smaller than the original documents, and that following a path p in the summary leads one to a set of node ids that corresponds to the nodes that would have been reached by following p in the original documents.

The graph summarization approach has proven to be very effective when applied to queries that examine the "coarse" structure of documents. For example, for many documents, a query //section/figure/title would be evaluated very efficiently by a graph summary index. In such an instance the graph summary functions as a kind of "path index" or "join index," and at query processing time the system can exploit the fact that the index building process has in effect precomputed a lot of the joins that would be time consuming in the inverted list-based approach.

Unfortunately, the graph summarization indexing approach is much less successful when we consider queries

on "values" or text words in the documents. This is roughly because any summary that retains enough detail to answer such queries has to be big (it has to encode a lot of details about specific values) so running queries over the summary will be no more efficient than running them over the original data.

Our contributions in this regard are:

- **Evaluating path expressions using structure indexes and inverted lists** (Section 3) We show how we can use structure indexes in conjunction with inverted lists to efficiently answer queries with both structure and value components. Our approach is to augment the inverted list entries with information derived from a structure index. Our query evaluation algorithm uses these modified entries to eliminate most inverted list joins.
- **Evaluation of these techniques** (Section 7) We have implemented our approach in the Niagara XML data management system [9]. Our preliminary experiments using Niagara demonstrate that we can derive substantial benefits by integrating the two forms of indexes.

While finding all documents or elements that satisfy a given path expression is a common use of path expression querying, users who specify keyword-based IR queries typically want just the k most relevant answers. To facilitate this kind of querying, we expand the IR notion of *relevance* to apply to path expression queries (in Section 4). Given a query, we rank all documents that match the query and return the top k documents in order of relevance, along with the specific elements that matched the query in each of these documents. The optimization challenge here, of course, is to try to find the top k answers without evaluating the entire query.

Our contributions here are:

- **Algorithm to Merge Ranked Inverted Lists** (Section 5) We adapt Fagin et al.'s Threshold Algorithm [11] to join ranked inverted lists. This is interesting because our setting poses novel challenges, as discussed in Section 4.2.
- **Using Structure Indexes for top-k Computation** (Section 6) The above algorithm is "instance optimal" across a broad class of algorithms. However, in our domain, the presence of additional access paths leads to new algorithms that are better on some instances. The above algorithm thus fails to be instance optimal in the presence of these new access paths. However, we show that a structure index can be used in conjunction with the ranked inverted lists to design a new algorithm that is instance optimal even in the presence of these access paths. We have implemented this algorithm in the Niagara system and present the results of preliminary experiments in Section 7.

2 Background

2.1 Data Model

Each XML document is a tree. An XML tree is a directed graph $G = (V_G, V_T, E_G, root, \Sigma_G, oid, label, ord)$. V_G is the set of element nodes while V_T is the set of text nodes, one per keyword in the XML document. E_T is the set of edges which are constrained to induce a spanning tree over $V_G \cup V_T$. Each edge in E_T is a parent-child edge. There is a distinguished node in V_G called the *root* with no incoming edges. Nodes in V_T have no outgoing edges, that is, they occur at the leaves of the tree. Nodes in $V_G \cup V_T$ are labeled through the *label* function. We assume that the labels of nodes in V_T are the respective keywords they represent and that they are distinct from those of nodes in V_G. The labels of nodes in V_T are placed in quotation marks to distinguish them. All nodes in $V_G \cup V_T$ are assigned unique ids through the *oid* function. Each node is assigned a unique *ordinal number*, through the *ord* function, which corresponds to its sibling position. We can define a total ordering on all nodes in $V_G \cup V_T$ by ordering parents before children and using the ordinal number between siblings. We refer to this as the *document order*. The document order corresponds to the order in which the data appears in the XML document.

Figure 1 is an example XML tree. This data represents one of the XQuery use cases available at [4]. The data represents an XML document that stores the contents of a book, in this case "Data on the Web". The book has a root book element along with tags for sections, figures, titles and paragraphs (p). These tags induce a tree structure on the document. The actual contents of the book appear at the leaf level of this tree. Some of these contents are omitted for clarity.

An XML database is a collection of XML trees/documents. The *oid*s are constrained to be unique across the whole database. The id of the root node of a document is the document id. The whole database consists of an artificial root node with the special label ROOT that has as its children the roots of each individual document. An example would be a database of books where each book is an XML document, like the one in Figure 1.

2.2 Path Expression Queries

A simple path expression has the form "$s_1\ l_1\ s_2\ l_2 \ldots s_k\ l_k$" where each l_i except l_k is a tag name, l_k is a tag name or keyword, and each s_i is either / or // denoting respectively parent-child and ancestor-descendant traversal.

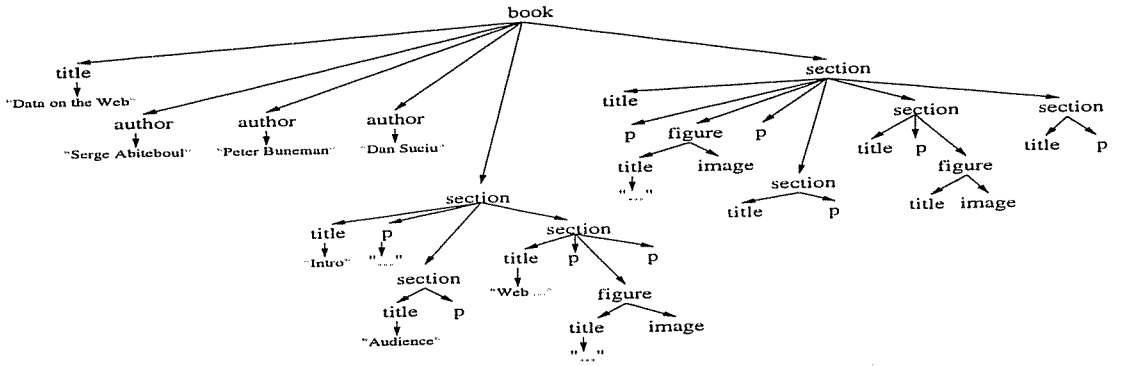

Figure 1: Sample data

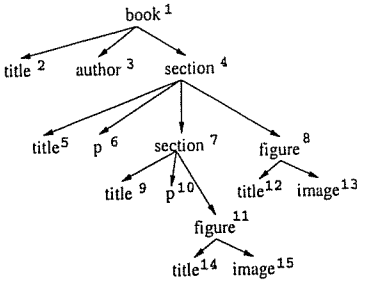

Figure 2: Example structure index

A branching path expression has the form "$s_1\ l_1[Pred_1]\ s_2\ l_2[Pred_2] \ldots\ s_k\ l_k[Pred_k]$" where each $Pred_i$ is an optional predicate, each l_i except l_k is a tag name, l_k is a tag name or keyword, and each s_i is either / or // denoting respectively parent-child and ancestor-descendant traversal. If l_k is a keyword, then $Pred_k$ must be absent. A predicate is a simple path expression.

The result is the set of all nodes that match the path expression query. This is standard notation for path expressions, with the exception that we allow the trailing label to be a keyword.

Some example queries on the data in Figure 1 are:

1. //section//title/"web"
2. //section[/title]//figure
3. //section[/title/"web"]//figure[//"graph"]

If a branching path expression has at least one keyword, we call it a *text query*. Otherwise, we call it a *structure query*. Queries 1 and 3 are instances of text queries while Query 2 is an instance of a structure query. The *structure component* of a text query TQ is the structure query $SQ(TQ)$ obtained by dropping all keywords from TQ. For instance, the structure component of Query 3 above is Query 2.

2.3 Structure Indexes

A structure index $I(G)$ for the data graph G corresponding to an XML database is another labeled, directed graph. The idea is to preserve all the *paths* in the data graph in the summary graph, while having far fewer nodes and edges. A structure index is used for query answering by associating an *extent* with each node in the index. In general, any partition of the element nodes defines a structure index where we (1) associate an index node with every equivalence class, (2) define the *extent* of each index node n, $ext(n)$, to be the equivalence class that formed it and (3) add an edge from index node A to index node B if there is an edge from some data node in $ext(A)$ to some data node in $ext(B)$. Henceforth, whenever we refer to a structure index, we mean an index obtained from a partition of the data nodes through the above construction. Thus, even a simple grouping of the data nodes by label defines a structure index. Each node A in the index has a unique identifier $id(A)$. Notice that a structure index indexes only the structural part of the XML database — it ignores the text nodes.

Figure 2 shows an example structure index. The numbers shown beside each node indicate the id of that node in the index. Each element node is associated with exactly one index node inducing a partition on the data nodes.

The *index result* of executing a path expression R on $I(G)$ is the union of the extents of the index nodes that match R. The extent mapping has the property that it is *safe*, that is, the result of any path expression R on G is contained in the result of R on $I(G)$. A structure index is said to be *precise* if the converse holds. For a particular path expression query Q, if the index result is equal to the result of Q on the data graph, then $I(G)$ is said to *cover* Q. A structure index that is precise covers all queries.

2.4 Inverted Lists

Several native XML database systems [9, 8] create inverted lists on tag names and keywords. Algorithms to effectively process queries using these lists have been proposed [21, 23].

We assume the following representation for inverted lists.

- For each element node n with tag t, there is an entry in the corresponding inverted list of the form $<docid, start, end, level, indexid>$. We denote $start$ as $n.start$ and likewise for the other fields.

- For each text node with label K, there is an entry in the corresponding inverted list of the form $<docid, start, level, indexid>$.

Here, *docid* refers to a unique document identifier and *level* is the depth of the node in the tree. The *start* and *end* numbers need to satisfy the following properties:

1. For each element node n, $n.start < n.end$.
2. If (element) node n_1 is an ancestor of element node n_2, then $n_1.start < n_2.start < n_2.end < n_1.end$.
3. If (element) node n_1 is an ancestor of text node n_2, then $n_1.start < n_2.start < n_1.end$.
4. If element nodes n_1 and n_2 are siblings and $ord(n_1) < ord(n_2)$, then $n_1.end < n_2.start$. A similar property holds when one or both of n_1 and n_2 are text nodes.

In order to integrate structure indexes with inverted lists, we add a new *indexid* field to the list entries. For a specific structure index I, the *indexid* field is set as follows.

- For an element node n, let the unique index node in whose extent n appears be N. Then, $n.indexid = id(N)$.
- For a text node n, let the unique index node in whose extent the parent of n appears be N. Then, $n.indexid = id(N)$.

For example, for the data shown in Figure 1, with first level section elements (that is, children of the root), we store an index id of 4. For the keyword "web" occurring under book/title, we store an index id of 2 corresponding to book/title in the index.

3 Evaluating Path Expression Queries

We first present a simple scenario to illustrate how we can integrate structure indexes and inverted lists. We then present the details of how a structure index can be used to convert a simple path expression query into an inverted list scan. For branching path expressions, it turns out that the number of joins to be performed can be reduced using a structure index. Finally, we show that even for simple path expressions, performing multiple joins can out-perform a scan. We introduce the notion of extent chaining to address this issue.

3.1 A Simple Example

Consider the following query over the data shown in Figure 1:

//section[figure/title/"graph"]

that asks for all sections that have a figure whose title contains the keyword "graph". Here, // refers to the ancestor-descendant separator while / refers to the parent-child separator in the XML tree.

Evaluating the above query over a native XML database system like Niagara [9] or Timber [8] would involve joining the inverted lists corresponding to the tag names section, figure and title, and the key-word "graph". Now suppose that we have a structure index on this data, for instance the 1-Index [18], which is shown in Figure 2.

Now consider the following evaluation strategy.

1. Execute the structure component //section[//figure/title] on the structure index to obtain a set of pairs of index ids corresponding to matching <section,title> pairs. In this case, this step would return $S = \{<4, 12>, <4, 14>, <7, 14>\}$.
2. Evaluate the join section[//"graph"] using the respective inverted lists, with the additional condition that a joining <section,"graph"> pair satisfies: the corresponding index id pair must be in S.

This strategy is correct since for any joining node pair $<n_s, n_w>$ (here, n_s is an element node with label section and n_w is a text node with label "graph"):

1. The fact that the parent of n_w has index id 12 or 14 means that n_w is under the path figure/title.
2. Since n_s has *some* path to n_w and since n_w is under figure/title, n_s satisfies the query.

Notice that we replace three joins with one, in the process incurring an index evaluation cost. The structure index is typically much smaller than the data. Hence, the evaluation using the structure index is likely to do well.

3.2 Simple Path Expressions

The algorithm for evaluating a simple path expression q using a structure index I is given in Figure 3. Steps 2-4 extract the structure component q' of q and check whether I covers q'. We assume that I comes with an interface to check this property. The algorithm uses I only if it covers q'. In this case, it evaluates q' on I to obtain a set S of index ids. If t is a tag name, then since I covers $q' = q$, Step 11 returns exactly the entries matching q.

If t is a keyword and *sep* is /, then for each entry e returned in Step 11, the following holds: $e.indexid \in S$ which means that the parent of e matches $q' = p$. Hence e matches q. The algorithm handles the case when *sep*

procedure evaluateSPEWithIndex(q, I)
/* evaluate simple path expression q using index I */
begin
1. Let $q = p\ sep\ t$
2. if (t is a keyword) then $q' = p$
3. else $q' = q$
4. if (I does not cover q') then
5. evaluateWithoutIndex(q)
6. Evaluate q' on I
7. Let S be the set of indexids returned
8. if (t is a keyword and sep is //) then
9. foreach ($i \in S$) do
10. put all descendants of i in S
11. Scan the inverted list for t returning only those entries e where $e.indexid \in S$

end

Figure 3: Using structure index for simple path expression

is // by adding the (index) ids of descendants of all (index) nodes matching p (Steps 8-10).

3.2.1 Branching Path Expressions

A branching path expression consists of multiple simple path expressions. We adapt the solution for simple path expressions to address each individual branch and then *join* appropriate lists.

We discuss the evaluation algorithm for branching path expression queries with one predicate. These ideas extend to generic branching path expressions in a straightforward manner. Queries with one predicate can be represented as $p_1[p_2\ sep\ t]p_3$ where p_1,p_2 and p_3 are simple structure expressions, sep is / or // and t is a keyword. Examples of queries of this kind are:

Q1 //section[/section/title/"web"]/figure/title
Q2 //section[/section//title/"web"]/figure/title
Q3 //section[/section/title/"web"]//figure/title
Q4 //section[/section/title//"web"]/figure/title

We assume that the structure index covers p_1, $//p_2$ and $//p_3$. Depending on the presence of // in p_2, p_3 or sep, we get the following cases.

Case 1: None of p_2, p_3 and sep contains //, as in Q1.
Case 2: p_2 contains //, as in Q2.
Case 3: p_3 contains //, as in Q3.
Case 4: sep is //, as in Q4.

Cases 2,3 and 4 are not disjoint.

In addition to the usual parent-child and ancestor-descendant join, we make use of the level numbers in the inverted list entries to perform *level* joins. For instance, section/^{2}title returns all title elements that are grand-children of a section element. In general, we use the notation $e_1/^d e_2$ to denote a binary level join. This can be trivially implemented by comparing level numbers during an ancestor-descendant check.

procedure evaluateWithIndex(q, I)
begin
1. Let $q = p_1[p_2\ sep\ t]p_3$
2. if (I does not cover p_1 or $//p_2$ or $//p_3$) then
3. evaluateWithoutIndex(q)
4. Let l_1, l_3 be the trailing tag names of p_1, p_3 respectively
5. Let $d2$ = number of tag names in $p_2 + 1$
6. Let $d3$ = number of tag names in p_3
7. Let $p_2' = /^{d2}t$
8. Let $p_3' = /^{d3}l_3$
9. Evaluate $q' = p_1[p_2]p_3$ on I
10. Let indexTriplets = $\{< i_1, i_2, i_3 >: i_1, i_2, i_3$ match l_1, l_2, l_3 respectively in the evaluation of q' on $I\}$
11. if (sep is //) then /* matches case 4 */
12. foreach ($< i_1, i_2, i_3 > \in$ indexTriplets) do
13. foreach (i_2' descendant of i_2) do
14. add $< i_1, i_2', i_3 >$ to indexTriplets
15. $p_2' = //t$
16. if (q matches case 2) then
17. skipJoins2 = true
/* verify if we can use index to skip joins in p_2 */
18. foreach ($< i_1, i_2, i_3 > \in$ indexTriplets) do
19. skipJoins2 = exactlyOnePath(i_1,i_2)
20. if (skipJoins2 is true) then $p_2' = //t$
21. else $p_2' = p_2\ sep\ t$
22. if (q matches case 3) then /* symmetric to case 2 */
23. skipJoins3 = true
/* verify if we can use index to skip joins in p_3 */
24. foreach ($< i_1, i_2, i_3 > \in$ indexTriplets) do
25. skipJoins3 = exactlyOnePath(i_1,i_3)
26. if (skipJoins3 is true) then $p_3' = //l_3$
27. else $p_3' = p_3$
28. if (skipJoins2 is false) then
29. foreach ($< i_1, i_2, i_3 > \in$ indexTriplets) do
30. $i_2 = \top$
31. if (skipJoins3 is false) then
32. foreach ($< i_1, i_2, i_3 > \in$ indexTriplets) do
33. $i_3 = \top$
34. Perform the join $l_1[p_2']p_3'$ using indexTriplets and return the results

end

procedure exactlyOnePath(i_1,i_2)
/* i_1 and i_2 are nodes in I */
/* returns true if there is exactly one path from i_1 to i_2 */
begin
1. Go backwards from i_2 to i_1 in I
2. Let p be a path from i_1 to i_2
3. if (any node in p has $>$ one in-coming edge in I) then
4. if (sources of > 1 are reachable from i_1) then
/* found 2 paths from i_1 to i_2 */
5. return false
6. if (i_1 is part of a cycle) then
7. return false
8. return true

end

Figure 4: Evaluation algorithm for branching path expressions using structure index

In Section 3.2, we saw how we can augment the scan of an inverted list to incorporate a set of indexids. Using this idea, we were able to convert a simple path expression query into a scan of a single list. We generalize this approach to inverted list joins as follows. For a 2-way join, we use a set S of indexid pairs obtained using the structure index to filter the result of the join so that only those pairs of entries whose indexids match some pair in S are returned. For n-way joins, we use a set S of n-tuplets of indexids. We use the special entry $\top$ for an indexid to denote that any value is a match.

The algorithm for evaluating a path expression query using a structure index is shown in Figure 4. We explain it by discussing how it handles Cases 1 and 2 above. Cases 3 and 4 can be similarly handled.

Consider Q1. Let the structure index I be the one shown in Figure 1. I is applicable since it covers the three expressions //section, //section/title and //figure/title. By evaluating the structure component of the query, //section[/section/title]/figure/title on I, we obtain a set S of triplets of ids of index nodes matching section, section/title and figure/title nodes (steps 9 and 10). In this case, $S = \{< 4, 9, 12 >\}$. We then evaluate the join //section[/^{3}"web"]/^{2}title using S. This strategy is correct since if $<n_s, n_w, n_t>$ is a node-triplet returned finally (with corresponding labels section, "web" and title):

1. n_s matches //section, n_w matches //section/title/"web" and n_t matches //figure/title.
2. n_s is the great grand-parent of n_w (due to a level difference of 3), so $<n_s, n_w>$ matches //section[/section/title/"web"].
3. n_s is the grand-parent of n_t (level difference of 2), so $<n_s, n_w, n_t>$ matches Q1.

We now move on to Case 2. Consider Q2. The main difference from Case 1 is that there is a // as part of the predicate which means that, for Q2, the distance between a section node and a "web" node is not known in advance. Suppose evaluating the structure component of Q2 on I returns a set of triplets S. Now, the idea is to check if we can skip the section//title join in the predicate. In order to replace e_1 =//section[/section//title/"web"] with e_2 =//section[//"web"] using S, we need to verify the following. If an entry s (corresponding to node n_s) in the inverted list for section and an entry w (corresponding to node n_w) in the inverted list for "web" satisfy e_2 and some triplet $< i_1, i_2, i_3 > \in S$, then there must actually be a path from n_s to n_w matching /section//title/"web". We ensure this by checking that there is exactly one path in the structure index from i_1 to i_2 through the function exactlyOnePath(i_1,i_2). Now, we know that there is *some* path p/"*web*" from n_s to n_w because of the containment check. By the property of structure indexes, there is a path matching p from i_1 to i_2. Also, since $< i_1, i_2, i_3 > \in S$, there is a path p' matching section//title from i_1 to i_2. But since there is exactly one path from i_1 to i_2, $p = p'$. Hence, we can skip the joins. As for the /figure/title join, since there is no // separator, it can be replaced with /^{2}title (as in Case 1). Putting this together, we evaluate the join //section[//"web"]/^{2}title using S.

3.3 Extent Chaining

In the above algorithm, we attempt to skip joins whenever possible using the structure index. As we will see next, it turns out that skipping joins is not always beneficial. We introduce the notion of extent chaining to address this deficiency.

Consider the query q =//figure/title. Using I, the algorithm converts q into a scan on the title inverted list with $S = \{< 12 >, < 14 >\}$. Suppose a document has 100 titles. In this case, the scan would examine the 100 title entries. Suppose only 10 occur directly under a figure, the other 90 being section titles. In [5], the authors introduce algorithms to make use of B-tree indices on the inverted lists while performing containment joins. The algorithm skips those parts of the inverted lists that do not participate in the join. Depending on the document structure, the join could return the 10 figure/title nodes by examining far fewer than 100 entries. Next, we discuss how to address this problem using the structure index.

The algorithm in [5] uses the fact that title is constrained to be under figure to skip irrelevant parts of the title inverted list. Observe that we can achieve a similar effect using the set of indexids corresponding to //figure/title. This is done by chaining all title entries based on indexids. That is, each entry has a pointer to the next entry in the same document with the same indexid. We refer to this as *extent chaining*. Now the inverted list entry for an element and keyword has an additional *next* field for this pointer.

The scan of an inverted list is modified to take advantage of extent chaining as follows. The algorithm is shown in Figure 5. In step 3, we obtain the first entry in a list corresponding to a given indexid. We maintain a directory for this purpose. If the database contains only one document, for instance, then the structure index itself can store this information.

Generalizing this approach to joins of inverted lists, we pass the projection of the appropriate column of S (set of indexid n-tuples for an n-way join) to the corresponding scan.

```
procedure scanWithChaining(L, S)
    /* returns entries in list L with indexid∈ S */
begin
1.  currEntries = φ
2.  foreach (id ∈ S) do
3.    add first entry in L with indexid id to currEntries
4.  while (currEntries ≠ φ) do
5.    minEntry = entry with minimum
                start number in currEntries
6.    get entry e in L corresponding to minEntry
7.    delete minEntry from currEntries
8.    if (minEntry.next ≠ NULL) then
9.      add minEntry.next to currEntries
10.   output e
end
```

Figure 5: Scan with extent chaining

4 Ranked IR-Style Path Expression Queries

We now consider how to support information retrieval style relevance-based querying over a corpus of XML documents. We first define the class of queries we consider and describe the associated relevance semantics. We then discuss the challenges involved in pushing down top k computation.

4.1 Query Language and Ranking Metric

In the classical information retrieval world view, each document is a bag of words and the query is also modeled as a bag of words. In our context, the database is a collection of XML documents that are modeled as *trees*. Hence, we expand the class of queries beyond keyword specification to allow simple structure constraints. More precisely, a relevance query is a bag of simple path expressions.

We next examine one way of defining the relevance of a document with respect to a relevance query. This definition is consistent with various proposals made for this purpose in the information-retrieval literature [12, 15, 19]. Let D be an XML document and p be a simple path expression query. The *term frequency* of p in D, $tf(p, D)$, is defined to be the number of (distinct) nodes in D that match p. As a special case, if p is $//t$ where t is a tag name or keyword, we refer to $tf(p, D)$ as the term frequency of t. The relevance of D with respect to p is calculated through a ranking function $R(p, D)$ that satisfies the following property: for path expressions p_1 and p_2, $tf(p_1, D) < tf(p_2, D) \Leftrightarrow R(p_1, D) < R(p_2, D)$.

Let $Q = \{p_1, \ldots, p_l\}$ be a bag of simple path expressions. Using the ranking function, we can talk about the relevance of document D for each p_i. The relevance of D with respect to Q is computed by combining all $R(p_i, D)$ through a merging function $MR(R(p_1, D), \ldots, R(p_l, D))$ (Merge Relevances). We require that this merging function be *monotonic* [11], that is, for documents D_1 and D_2, if $R(p_i, D_1) \geq R(p_i, D_2)$ for each i from 1 to l, then $MR(R(p_1, D_1), \ldots, R(p_l, D_1)) \geq MR(R(p_1, D_2), \ldots, R(p_l, D_2))$. Any pair of ranking and merging functions that satisfy the above properties is sufficient for our algorithm to work.

4.2 Challenges in Pushing Down Top k Computation

The main problem in this domain is to try to find the top k answers without evaluating the entire query. In order to push down the top k computation, we need access paths based on relevance. We assume that for each tag name (keyword) t, there is an additional inverted list $rellist(t)$ where the entries within a document are in document order and the inter-document order is in descending order of relevance of t ($R(t, D)$).

Fagin et al. proposed the threshold algorithm (TA) to merge ranked lists in middleware [11]. There are two main differences in our setting.

- When we join two inverted lists, the relevance of the result is not "monotonic" in the relevance of the inputs. In other words, suppose we are evaluating a//b. If we were to directly apply the threshold algorithm, then we need the following property: for documents d1 and d2, if R(a,d1)>R(a,d2) and R(b,d1)>R(b,d2) then R(a *sep* b,d1)>R(a *sep* b,d2). This is not true in our scenario.

- TA is a middleware algorithm and is provably optimal under certain assumptions. Our focus is on the XML database *server* where additional access paths, like the original inverted lists, are available. These access paths violate the assumptions under which TA is proved to be optimal.

We next explore how each of these differences can be handled.

5 Adapting TA to Inverted List Joins

We present the details for two-way join queries. The adaptation for more joins is straight-forward. Consider the path expression a *sep* b. The algorithm for this case, compute_top_k, is given in Figure 6. For steps 10 and 15, we can use any standard algorithm that merges two inverted lists [21, 23].

The procedure compute_top_k executes a *sep* b on a per-document basis in the process maintaining the top k documents (based on relevance) among the documents processed so far in the set topKresults. When it realizes that none of the future documents can be part of the top k, it stops processing and returns the results. This termination condition is shown in Step 7. The maximum relevance any future document can have is the total number of b entries in the current document in ListB. If the

```
procedure compute_top_k(k,a,sep,b)
    /* query is: a sep b; sep is / or // */
begin
1.  ListA = rellist(a) /* relevance list for a */
2.  ListB = rellist(b) /* relevance list for b */
3.  topKresults = φ
4.  mintopKrank = 0
5.  while (more entries in both ListA and ListB) do
6.    currDocB = next document in ListB
7.    if ((R(b,currDocB) <= mintopKrank) and
        (number of documents in topKresults is k)) then
8.      break
9.    if (currDocB ∉ topKresults) then
10.     Evaluate a sep b on currDocB
11.     Let the result be currDocResult
12.     Add currDocResult to topKresults
13.   currDocA = next document in ListA
14.   if (currDocA ∉ topKresults) then
15.     Evaluate a sep b on currDocA
16.     Let the result be currDocResult
17.     Add currDocResult to topKresults
18.   Retain only top k documents in topKresults
19.   Set mintopKrank appropriately
20. return topKresults
end
```

Figure 6: top k algorithm for 2-way join

latter value is smaller than the relevance of the k^{th} document in topKresults, then no more documents need to be processed since the list is ordered by relevance. In addition, if we have seen all entries in either list, then the join terminates. This is so since we have executed the join for all documents containing both a and b.

The main difference from the original threshold algorithm is the use of R(b,currDoc) in Step 7 above. Also, unlike the original threshold algorithm, we do not assume that each document appears in every list. We handle this through the condition for the while loop in Step 5.

For a generic simple path expression query Q, we modify compute_top_k by using the list corresponding to the result node of Q to define the terminating condition like in Step 7 above, and evaluating Q for each document accessed, using any standard query evaluation algorithm [3, 21, 23]. The details are omitted.

5.1 Instance Optimality

In [11], the notion of instance optimality is introduced and it is shown that the threshold algorithm is instance optimal among a certain class of algorithms. Similar results apply in our context. We use the following terminology from [11] to formalize this claim.

We consider the following modes of access to the relevance lists. For a particular list L, we can obtain the entries for the next document in relevance order — this corresponds to a *sorted access* to that document. Alternatively, we can specify a document id and ask for all entries pertaining to it. This is a *random access* to that document. Either access to a document returns *all* entries in that document. An algorithm to compute the top k documents is said to make a *wild guess* if it makes a random access on list L for a document id without having seen it under sorted access under some (possibly other) list.

We now recall the notion of instance optimality [11]. Let $\mathcal{A}$ be a class of algorithms, and let $\mathcal{D}$ be a class of legal inputs to the algorithms. We assume that we are considering a particular non-negative cost measure $cost(A, D)$ of running algorithm A over input D. We say that an algorithm $B \in \mathcal{A}$ is instance optimal over $\mathcal{A}$ and $\mathcal{D}$ if for every $A \in \mathcal{A}$ and $D \in \mathcal{D}$, we have: $cost(B, D) = O(cost(A, D))$. In other words, there are constants c, c' such that $cost(B, D) \leq c \times cost(A, D) + c'$ for every choice of A and D. We note that instance optimality is a stronger notion of optimality than worst-case, or even average-case optimality.

In our context, we define $cost(A, D)$ of running algorithm A over input D to be the number of document accesses, both sorted and random, by A across all lists. If a document is accessed on multiple lists, it is counted once per list. Similarly, if a document is accessed multiple times in the same list, it is counted once per access.

Theorem 1: *Let q be a simple path expression query. Let $\mathcal{D}$ be the class of all databases. Let $\mathcal{A}$ be the class of all algorithms that correctly find the top k documents (and corresponding nodes) for q over every database and that do not make wild guesses. Then, compute_top_k is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

5.2 Issues With Additional Access Paths

Recall that we have inverted lists sorted on document id in addition to lists in relevance order. Just as in Section 3.3, where we skip parts of an inverted list within a document using secondary indexes, it is possible to skip *documents* during a containment join over all documents. We illustrate this next with an example. Consider the simple path expression query q = a/b. Suppose the XML database has 201 documents with ids from 1 to 201. Let documents 1 to 100 have only a elements and documents 101 to 200 have only b elements. Let document 201 have an a element with child b. Consider the following algorithm for evaluating q.

1. Look at the first document in the two lists — 1 and 101.
2. Since the document ids are different, use the larger id (in this case, 101) to seek to the first document in the list for a with document id *greater than or equal to* 101.
3. The list for a is now positioned at document 201.
4. Since the document ids are still different, seek on the list for b to the first document with id $\geq$ 201.
5. Now both lists are positioned at 201.

```
procedure compute_top_k_with_sindex(k,q,sep,b)
      /* query is: q sep b; sep is / or //,
        q is a simple path expression */
begin
1.  ListB = rellist(b) /* relevance list for b */
2.  if (b is a tag name) then
      //evaluate q sep b on the structure index
3.    indexidList = list of ids of index nodes
            matching q sep b
4.  else /* b is a keyword */
5.    if (sep is /) then
6.      indexidList = list of ids of index nodes matching q
7.    else /* sep is // */
8.      indexidList = list of ids of index nodes matching q
            and their descendants in the structure index
9.  topKresults = φ
10. mintopKrank = 0
11. while (more entries in ListB) do
12.   currDoc = next document in ListB with at least
            one entry e such that e.indexid ∈ indexidList
            (use extent chaining)
13.   if ((R(b,currDoc) < mintopKrank) and
        (number of documents in topKresults is k)) then
14.     break
15.   currDocResult = {e : e ∈ ListB corresponding to
            currDoc and e.indexid ∈ indexidList}
            (use extent chaining)
16.   Add currDocResult to topKresults
17.   if (topKresults has k + 1 documents) then
18.     remove document with least relevance
19.   Set mintopKrank appropriately
20. return topKresults
end
```

Figure 7: top k algorithm using structure index

6. Perform the join over document 201.
7. Since there are no more documents on both lists, return.

This evaluation accesses only three documents. On the other hand, compute_top_k accesses all documents. The above algorithm performs efficiently on this instance due to the presence of a secondary index. Notice that in Step 3, the list for a is positioned at document 201 as a result of the random access in Step 2. But document 201 is not accessed through sorted access before this. This classifies as a wild guess and is not permitted in the class of algorithms considered in the instance optimality discussion.

We next show how we obtain an instance optimal algorithm even in the presence of these access paths. We use a structure index along with extent chaining for this purpose.

6 Instance Optimality Using A Structure Index

We show how structure indexes can be used to obtain an instance optimal algorithm even in the presence of these access paths. We first consider the case when the relevance query has a single path expression. We then extend our algorithm in Section 6.1 to the case when the relevance query is a bag of path expressions.

The evaluation of a simple path expression $Q = q\ sep\ b$ using a structure index I that covers it results in a scan on the inverted list of b with a set S of indexids. The algorithm for computing the top k documents in this case is shown in Figure 7. We modify the idea of extent chaining introduced in Section 3.3 to chain all entries in the relevance inverted lists with the same indexid even across documents. Thus, each entry has a pointer to the next entry with the same indexid even if it is not in the same document. We observe the following about this algorithm.

- Steps 2-8 initialize the indexidList appropriately depending on whether b is a tag name or keyword and whether *sep* is / or //.
- The terminating condition in Step 13 is similar to the one in the procedure compute_top_k.
- The evaluation of currDocResults in Step 15 (for a single document) can be performed using intra-document extent chaining described in Section 3.3.
- In Step 12, we use inter-document extent chaining to advance to the next document in ListB having at least one match for q *sep* b.

Implementation Note

When performing a scan using extent chaining, to get the next entry in the list (like in Step 5 in Figure 5), we might need to compare the next pointers of more than one entry and find which of them appears first in the relevance list. The relative position of two documents in a relevance list cannot be obtained by comparing their document ids. Hence, we introduce relevance document ids (reldocids). All documents appearing in a relevance list are assigned reldocids based on their order in the list. The next pointer of an entry contains the reldocid and start number of the next entry with the same indexid. Using the reldocids, we can compare the next pointers of more than one entry. An entry in the relevance list for a tag name is of the form: $<reldocid, start, end, level, indexid, docid, next\_reldocid, next\_start>$. An entry for a keyword is the same except for the absence of *end*. We emphasize that the reldocid is used only for extent chaining. In particular, when we talk about document ids, we refer to the unique document id that is common to a document across all lists.

Instance Optimality

In addition to the sorted and random access modes on the relevance lists, we allow sorted and random access on the inverted lists sorted on document id. We relax the wild

procedure compute_top_k_bag(k,p_1 sep_1 a,p_2 sep_2 b)
begin
1. ListA = rellist(a) /* relevance list for a */
2. ListB = rellist(b) /* relevance list for b */
3. Obtain indexidListA and indexidListB appropriately using the structure index
   //as in Steps 2-8 of compute_top_k_withsindex
4. topKresults = ϕ
5. mintopKrank = 0
6. while (more entries in either ListA or ListB) do
7. currDocA = next document in ListA, as per extent chaining
8. currDocB = next document in ListB, as per extent chaining
9. R_a = R(a,currDocA)
10. R_b = R(b,currDocB)
11. if ((MR(R_a,R_b) <= mintopKrank) and (number of documents in topKresults is k)) then
12. break
13. if (currDocA $\notin$ topKresults) then
14. Evaluate the two path expressions on currDocA
15. Let the result be currDocResult
16. Add currDocResult to topKresults
17. Do similarly for currDocB
18. Retain only top k documents in topKresults
19. Set mintopKrank appropriately
20. return topKresults

end

Figure 8: top k algorithm for 2 simple path expressions

guess definition to exclude the following: (1) random access on any list to first document with id $\geq$ a given id, and (2) random access on any list L to a document with reldocid obtained from the next field of an entry (in L). Cost is measured in the same way as in Section 5.1. In particular, the index evaluation cost is not counted for the purpose of this discussion.

Theorem 2: *Let q be a simple path expression query covered by structure index I. Let $\mathcal{D}$ be the class of all databases. Let $\mathcal{A}$ be the class of all algorithms that correctly find the top k documents (and corresponding nodes) for q over every database and that do not make wild guesses. Then, compute_top_k_with_sindex is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

6.1 Extension to Bag of Simple Path Expressions

We now extend the above algorithm to the case when the query is a bag of simple path expressions; intuitively, this corresponds to the class of IR queries with multiple keywords. Consider the evaluation of query $Q = \{p_1, p_2\}$. Using the structure index, we can convert each p_i to a scan on the appropriate relevance list. What remains now is to merge these relevance lists and apply the ranking function for Q (which merges the relevances of the p_i). This merge is similar to the merge algorithm in Figure 6. The algorithm is shown in Figure 8. Since the merging function is monotonic, this algorithm can easily be shown to be correct. This algorithm naturally extends when Q has more than two simple path expressions.

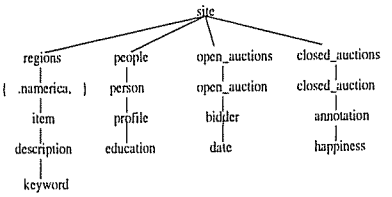

Figure 9: XMark Schema

We now show that this algorithm is instance optimal for an interesting class of bag queries, over the class of algorithms that do not make wild guesses, as defined above. A bag B of simple path expressions is defined to be disjoint if the trailing terms of no two simple path expressions in B are the same. For example, the bag {book/author,article/title} is disjoint while the bag {book/author,article/author} is not.

Theorem 3: *Let $Q = \{q_1, q_2, \ldots, q_l\}$ be a disjoint bag of simple path expression queries covered by structure index I. Let $\mathcal{D}$ be the class of all databases. Let $\mathcal{A}$ be the class of all algorithms that correctly find the top k documents (and corresponding nodes) for Q over every database and that do not make wild guesses. Then, compute_top_k_bag is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

7 Preliminary Experiments

We have implemented the above algorithms as part of the Niagara XML database system [9]. We present the results of some preliminary experiments that yield a sense for the efficacy of our techniques. We first present our results for evaluating branching path expression queries using structure indexes and inverted lists. We then move on to relevance queries. Our experiments are run on a Linux Workstation with 256MB of RAM. We use a 16MB buffer pool.

7.1 Evaluation of Branching Path Queries

We use the XMark XML-benchmark data [22] for this set of experiments. This data models an auction site. The element relationships relevant to this paper are shown in Figure 9. The tag names are self-explanatory. The data size is 100MB. The structure index we use is the 1-Index [18]. A study of how the choice of structure index impacts performance is future work. We report the performance results for four queries involving structure and value constraints based on warm buffer pool times. We measure the speedup, defined to be the ratio of the (execution) time taken in the absence of a structure index (inverted list join) to the time taken by our algorithm. In the presence of alternative query plans, we use the execution time corresponding to the best plan. Table 1 shows the queries and the respective speedups.

The main observations to be made from the above numbers are:

Query in English	Path expression	Speedup
Find occurrences of "attires" under item descriptions	//item/description//keyword/"attires"]	43.3
Find open auctions that had a bid in 1999	//open_auction[/bidder/date/"1999"]	6.85
Find the persons who attended Graduate school	//person[/profile/education/"Graduate"]	5.06
Find closed auctions where the happiness level was 10	//closed_auction[/annotation/happiness/"10"]	3.12

Table 1: Speedups Using Structure Index

Value of k	Speedup for Q1	# Documents Accessed by our algorithm	Speedup for Q2	# Documents Accessed by our algorithm
1	16.04	20	18.07	2
5	14.92	25	10.38	6
10	14.53	25	8.13	10
50	12.42	27	3.67	51
100	12.42	27	2.15	101
300	12.42	27	1.7	301

Table 2: Results for top k queries

- The benefit of using a structure index in conjunction with inverted lists is considerable with speedups of as high as about 43 times for simple path expressions and about 7 times for branching path expressions.

- The speedup obtained is dependent on the number of joins saved. At the extreme, if we remove all joins replacing them with a scan, then the speedup obtained is highest. Thus, for the first query above which is a simple path expression, the speedup obtained is highest.

7.2 Relevance Queries

We have implemented the compute_top_k_withsindex algorithm shown in Figure 7. Recall that this is an instance optimal algorithm for relevance queries consisting of a single (simple) path expression. We wish to study the benefit obtained through two aspects of this algorithm — the early termination condition and extent chaining. Consider a query $q = p//t$. In the scenario where t occurs in many documents but very few of these match q, extent chaining is likely to yield significant performance benefit. On the other hand, if t occurs in many documents and most of these occurrences match q, the early termination condition is likely to contribute.

To study this, we use NASA's public astronomy XML archive [2]. The data has 2443 XML documents with a total size of about 33MB. We consider two queries — Q1 and Q2 — that search for occurrences of a particular word "photographic" under two different paths p_1=keyword and p_2=dataset respectively. There are very few occurrences of "photographic" under keyword, while all occurrences are under dataset.

Table 2 shows the results of our experiment. For each value of k, we report the speedup obtained through our algorithm, measured as the ratio of the time taken to fully execute the query on the database to the time taken by our algorithm. We also report the number of documents accessed by our algorithm.

We observe first of all that there is a significant benefit to be obtained by pushing down the top k computation, instead of evaluating the query completely and then extracting the top k results. For Q1, notice that the number of documents accessed by our algorithm varies very little with k. This indicates that the benefit is chiefly through extent chaining. On the other hand, for Q2, the number of documents accessed increases linearly with k, showing the role played by the early termination condition.

8 Related Work

Several methods have been proposed for processing queries over graph-structured XML data. These methods can be classified into two broad classes. The first involves graph traversal where the data graph is traversed using the input query [6, 17]. The other involves information-retrieval style processing using inverted lists [3, 21, 23]. Methods have been proposed to optimize queries in the presence of both these alternatives [6, 8, 17].

In this framework, structure indexes such as the ones proposed in [13, 16, 18] have primarily been used as a substitute for graph traversal [17]. However, to the best of our knowledge, no published work has addressed how to integrate structure indexes with information retrieval style inverted list processing. This is the focus of our paper.

Several proposals have been made for ranked search over a corpus of document databases combining keyword and structure components [15, 19]. Recently, in [12], a query language is proposed to integrate information retrieval related features such as weighting, ranking and relevance-oriented search into XML queries.

The focus of our paper is not on defining the best query model over XML documents. Instead our interest is in understanding how we can efficiently push down top k computation and the role of structure indexes in this. Several previous projects [1, 7, 10, 11, 14] have dealt with supporting keyword search over structured databases. Our query language permits a structural component in addition to keyword specification. We examine how structure indexes help in handling the structural component.

9 Conclusions

We presented methods of integrating structure indexes and inverted lists. By appropriately augmenting inverted list entries, we showed how inverted list joins could be replaced with an index navigation when evaluating branching path queries. Our preliminary experiments on the Niagara native XML database system showed the efficacy of this approach.

Throughout our discussion, we assumed that an XML document has two parts — one that is summarized by the structure index and one that is not. We used element nodes and text nodes to identify these parts. There can be several ways of defining these parts. For instance, the values of some text nodes can be captured in the structure index by treating them as tag names. The techniques presented in this paper are applicable irrespective of how we arrive at these two parts. However, this paper is not about *how* we define these parts. This is an interesting area for future work. Other such areas include looking at the tradeoffs involved in picking a structure index and integrating multiple structure indexes with inverted lists.

We also considered the evaluation of top k queries over XML documents. We showed how the augmented "relevance" inverted lists combined with adaptations of the Threshold algorithm proposed by Fagin et al. yields instance optimal algorithms for pushing down top k computation. In our context, the ranking function is non-monotonic and there are additional access paths available. Using a structure index, we were able to successfully adapt the Threshold algorithm preserving the optimality properties. While we presented algorithms for tree structured data, they can be extended to work for graph-structured data. Several avenues remain for future work. For instance, incorporating proximity in the merge function and the problem of running structured queries over hyper-linked XML documents remain open.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of ICDE*, 2002.

[2] XML Astronomy Archive at NASA. XML astronomy archive at NASA. `http://xml.gsfc.nasa.gov/archive`.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD*, 2002.

[4] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A query language for XML. World Wide Web Consortium, `http://www.w3.org/TR/xquery`, Feb 2000.

[5] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of VLDB*, 2002.

[6] A. Halverson et. al. Mixed mode XML query processing. In *Concurrent Submission to VLDB*, 2003.

[7] G. Bhalotia et. al. Keyword searching and browsing in databases using BANKS. In *Proceedings of ICDE*, 2002.

[8] H.V. Jagadish et. al. TIMBER: A native XML database. *VLDB Journal*, 2003.

[9] J.F. Naughton et. al. The Niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2), 2001.

[10] R. Goldman et. al. Proximity search in databases. In *Proceedings of VLDB*, 1998.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of PODS*, 2001.

[12] N. Fuhr and K. Grobjohann. XIRQL: A language for information retrieval in XML documents. In *Proceedings of SIGIR*, 2001.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, 1997.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *(to appear in) Proceedings of SIGMOD*, 2003.

[15] M. Hearst and C. Plaunt. Subtopic structuring for full-length document access. In *Proceedings of SIGIR*, 1993.

[16] R. Kaushik, P. Bohannon, J.F. Naughton, and H.F. Korth. Covering indexes for branching path queries. In *Proceedings of SIGMOD*, 2002.

[17] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, 1999.

[18] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[19] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4), 1997.

[20] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill, 1983.

[21] D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.

[22] Xmark: The xml benchmark project. `http://monetdb.cwi.nl/xml/index.html`.

[23] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.