On the Integration of Structure Indexes and Inverted Lists
Paper Id: 570

ABSTRACT

Several methods have been proposed to evaluate queries over a native XML DBMS, where the queries specify both path and keyword constraints. These broadly consist of graph traversal approaches, optimized with auxiliary structures known as structure indexes; and approaches based on information-retrieval style inverted lists. However, no published literature addresses methods of combining structure indexes and inverted lists. We bridge this gap by proposing a strategy that combines the two forms of auxiliary indexes and a query evaluation algorithm for branching path expressions based on this strategy. Our technique is general and applicable for a wide range of choices of structure indexes and inverted list join algorithms. Our experiments over a native XML DBMS show the benefit of integrating the two forms of indexes. We also consider algorithmic issues in evaluating path expression queries when the notion of relevance ranking is incorporated. 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. For example, consider the query //section[/figure/title/"Graph"]. This query looks for the keyword "Graph" (its keyword component) appearing at the end of a sequence of structural containments //section/figure/title (its structural component) and returns the matching sections.

Several methods have been proposed for processing path expressions over graph/tree-structured XML data. These methods can be classified into two broad classes. The first involves graph traversal where the input query is evaluated by traversing the data graph [21, 31] or some compressed representation [4, 9, 39]. The other class involves information-retrieval style processing using inverted lists [8, 11, 25, 28, 37, 41, 45]. Methods have been proposed to optimize queries in the presence of both these alternatives [21, 24, 31]. In this framework, *structure indexes* [12, 18, 26, 27, 32] have been proposed to be used as a substitute for graph traversal [31]. These structure indexes are 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 structure index. Unfortunately, the structure 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. On the other hand, 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 can be expensive. To the best of our knowledge, no published literature addresses the problem of combining these two forms of auxiliary indexes.

This paper bridges this gap by proposing a strategy that combines structure indexes and inverted lists and a query evaluation algorithm for branching path expressions based on this strategy. Our algorithm does not assume any specific property of these indexes and is applicable for a wide range of structure indexes and inverted list join algorithms. Our contributions in this regard are:

- **Evaluating path expressions using structure indexes and inverted lists** (Section 3) We augment the inverted list entries with information derived from a structure index and propose a query evaluation algorithm that uses these modified entries to potentially eliminate most inverted list joins.
- **Evaluation of these techniques** (Section 6) We have implemented our approach in a native XML data management system [43]. Our experiments using this system 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. Several proposals have been made to incorporate the IR notion of *relevance* to XML queries [2, 15, 19, 30, 35]. As described in [16], XML search tasks can be divided into Content-Only (CO) tasks where XML documents are searched only using keywords, and Content-and-Structure (CAS) tasks where both structure and content is queried. Techniques such as XRank [19] deal with the CO space. In the CAS space, several approaches such as in [2, 15, 30] have been considered. However, the features of appropriate query languages are yet to be clearly identified.

In this paper, we focus on a subclass of CAS queries consisting of simple path expressions. We study algorithmic issues in integrating structure indexes with inverted lists for the evaluation of these queries, where we rank all documents that match the query and return the top k documents in order of relevance. We allow a broad class of relevance functions (Section 4.1) that covers the standard tf-idf notion of ranking and propose instance-optimal [14] methods of pushing down top k computation by combining the forms of indexes. Our approach is based on Fagin et al.'s Threshold Algorithm (TA) [14]. Our setting poses novel challenges (Section 4.2), since the ranking function we allow is not necessarily monotonic [14]. Also, unlike TA which is a middleware algorithm, our focus is on the database *server* where additional access paths are available. This violates the assumptions under which TA is shown to be instance optimal. Our contributions here are:

- **Algorithm to Merge Ranked Inverted Lists** (Section 4.3): We adapt Fagin et al.'s Threshold Algorithm [14] to join ranked inverted lists to evaluate a single path expression. The technical challenge is due to the fact that the ranking function we use is not monotonic, as required by the algorithms in [14].
- **Using Structure Indexes for top-k Computation** (Section 5): In our domain, the presence of additional access paths leads to new algorithms that are better on some instances of the problem. 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 extend this algorithm to the case when the query is a bag of simple path expressions (Section 5.1), in which case, it is instance optimal for a broad subclass of ranking functions allowed. We present the results of our experiments in Section 6.

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 [10]. 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. If l_k is a keyword, the simple path expression is called a *simple keyword path expression*.

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", which returns all occurrences of the keyword "web" under the path //section//title.
2. //section[/title]//figure
3. //section[/title/"web"]//figure[//"graph"]

If a branching path expression has at least one keyword,

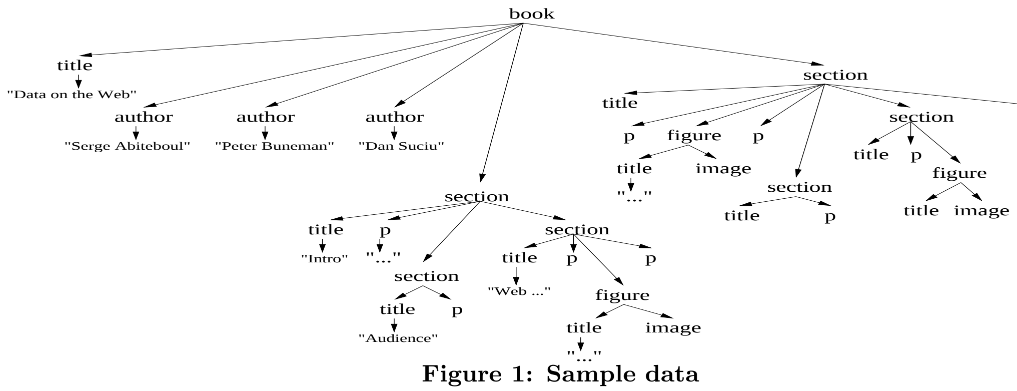

Figure 1: Sample data

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

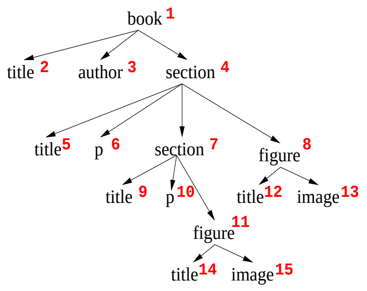

Figure 2: Example structure index

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 the result of any path expression R on G is contained in the result of R on $I(G)$. 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.

2.4 Inverted Lists

Several native XML database systems [24, 43] create inverted lists on tag names and keywords. Algorithms to effectively process queries using these lists have been proposed [37, 45]. 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.

Path expressions can be evaluated by joining inverted lists [37, 45]. In order to make these joins more efficient, auxiliary indexes have been proposed [8, 11, 21]. For example, in Niagara [21], B-Trees are used to skip parts of the inverted lists during query processing. We denote the algorithm that joins inverted lists to evaluate a path expression p as $IVL(p)$. We use IVL as a subroutine in our algorithm. Any of the published techniques to join inverted lists [8, 11, 25, 28, 37, 45] can be used for this procedure.

3. EVALUATING PATH EXPRESSION QUERIES

We first describe how we modify inverted lists to integrate them with structure indexes. Next, we present an example scenario to illustrate our query evaluation algorithm that uses this modification. We then present the details of our algorithm first for simple path expressions and next for branching path expressions. Finally, we show that there are cases when inverted list joins can out-perform a scan. We introduce the notion of extent chaining to address this issue.

3.1 Integrating Structure Indexes with Inverted Lists

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.

3.2 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".

Evaluating the above query over a native XML database system like Niagara [34] or Timber [24] 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 [32], which is shown in Figure 2. This index covers all simple path expressions.

A straight-forward method of using this index would be to use it only for those parts of the path expression that are directly covered by it. For instance, we could select only that part of the figure list that is under section, by using the index ids for //section//figure. This would have some benefit since fewer figure elements participate in the join. However, we can do even better as illustrated by 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.

3.3 Simple Path Expressions

```
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.     use $IVL(q)$ to evaluate without structure index
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

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

3.3.1 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 grandchildren 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.

In Section 3.3, 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. Notice that for any inverted list join algorithm IVL, the modification described above is straight-forward.

We explain our algorithm by discussing how it handles Cases 1 and 2 above. Cases 3 and 4 can be similarly handled. The detailed algorithm is omitted for lack of space.

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. In this case, $S = \{< 4, 9, 12 >\}$. We then evaluate the join //section[$/^3$"web"] $/^2$title using IVL with 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. 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 IVL with S.

3.4 Extent Chaining

In our algorithm described in Section 3.3.1, 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. In [11], the authors introduce algorithms to make use of B-tree indexes on the inverted lists while performing containment joins. The algorithm does not examine those parts of the inverted lists that do not participate in the join. Depending on the document structure, the join could return the figure/title nodes by examining far fewer than the total number of title entries. For example, suppose a document has 100 titles of which only 10 occur directly under a figure, the other 90 being section titles. In this case, the scan would examine the 100 title entries, whereas algorithms using B-Tree indexes could examine as few as 10 title entries in the best case.

In the algorithm described in the previous section, we attempt to skip joins whenever possible. As we just saw, however, joins could actually restrict the computation and make it more efficient. Next, we discuss how to adapt our algorithm to address this problem. The algorithm in [11] uses the fact that title is constrained to be under figure to ignore 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 4. 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. In Section 6, we discuss the tradeoff between performing a linear scan and using an extent chain.

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.

4. RANKED IR-STYLE PATH QUERIES

We now consider the role of structure indexes in supporting information retrieval style relevance-based querying over a corpus of XML documents. We first define the class of queries we consider and describe the

```
procedure scanWithChaining($L$, $S$)
    /* returns entries in list $L$ with indexid$\in S$ */
begin
1.  currEntries = $\phi$
2.  foreach ($id \in S$) do
3.      add first entry in $L$ with indexid $id$ to currEntries
4.  while (currEntries $\neq \phi$) 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 $\neq NULL$) then
9.          add $minEntry.next$ to currEntries
10.     output $e$
end
```

Figure 4: Scan with extent chaining

associated relevance semantics we allow. We next discuss the challenges involved in pushing down top k computation. Finally, we study the limitations of a straightforward extension of the Threshold Algorithm [14].

4.1 Query Language and Ranking Metric

Several proposals have been made to incorporate the IR notion of *relevance* to XML queries [2, 15, 19, 30, 35]. As described in Section 1, several of these proposals consider Content-and-Structure (CAS) tasks where both structure and content is queried. However, the features of appropriate query languages are yet to be clearly identified. The goal of this paper is not to define the best relevance metric over XML data, instead it is to study algorithmic issues in merging structure indexes with inverted lists in relevance-based computation. For this purpose, we consider a subclass of CAS queries.

We define a relevance query to be a bag of simple keyword path expressions, analogous to the "bag of words" query model in information retrieval. Thus, we allow simple structural specification in addition to keywords.

We next examine our model for relevance computation. Let D be an XML document and p be a simple keyword path expression query. The relevance of D to p is computed using a non-negative ranking function $R(p, D)$. For a bag of simple keyword path expressions, $Q = \{p_1, \ldots, p_l\}$, 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 non-negative merging function $MR(R(p_1, D), \ldots, R(p_l, D))$ (Merge-Relevance).

The ranking function R must be consistent with term frequency (tf). 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. $R(p, D)$ must satisfy the following property: for path expressions p_1 and p_2, $tf(p_1, D) < tf(p_2, D) \Leftrightarrow R(p_1, D) < R(p_2, D)$. We also require that if $tf(p, D)$ is 0, then $R(p, D) = 0$. We refer to the above as the *tf-consistency* property.

The merging function MR must be *monotonic* [14], 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 $MR(R(p_1, D_1), \ldots, R(p_l, D_1)) \geq MR(R(p_1, D_2), \ldots, R(p_l, D_2))$. We also require that for document D, if each $R(p_i, D)$ is 0, then $MR(R(p_1, D), \ldots, R(p_l, D)) = 0$. Any pair of ranking and merging functions that satisfy the above properties is permitted. Note that one particular example of a merge-rank function is a weighted sum of the individual ranks. Here, the weights could be inverse document frequencies (idf). Hence, the above definition of relevance permits the traditional IR notion of tf-idf based ranking.

4.1.1 Extending Ranking to Include Proximity

We can extend the relevance metric to account for keyword proximity when the input is a bag of simple keyword path expressions. For this purpose, we modify the relevance computation by multiplying the merge-rank function by a proximity function ρ. Thus, for a bag of path expressions $\{p_1, \ldots, p_l\}$, the relevance of a document D is measured as $MR(R(p_1, D_1), \ldots, R(p_l, D_1)) \times \rho(D, p_1, \ldots, p_l)$. We assume that the value of the proximity function lies in the range $[0, 1]$. Note that we do not assume anything about what the notion of proximity is: it could be measured by merely treating the document as a text document and using any standard IR notion of proximity, or could reflect the tree structure of the document by assigning a higher weight if there is a deeply nested element that contains all the keywords. A relevance function is said to be (1) *well-behaved* if R is tf-consistent, MR is monotonic and $\rho \in [0, 1]$, (2) *proximity-sensitive* if it is well-behaved and ρ is not identically 1. Note that for a single simple keyword path expression, MR and ρ are not applicable, instead relevance is computed using the ranking function R.

4.2 Optimizing 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 [14]. 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.

4.3 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 5. For steps 10 and 15,

```
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 = $\phi$
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 $\notin$ 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 $\notin$ 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 5: Top k algorithm for 2-way join

we can use any standard algorithm that merges two inverted lists [8, 11, 25, 28, 37, 45].

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 relevance of the current document in ListB. If the 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 keyword 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 [8, 11, 25, 28, 37, 45]. The details are omitted for lack of space.

4.4 Instance Optimality

In [14], 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 [14] 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* [14] 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 [14]. 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. Computing the relevance of a document is counted as one document access. 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.

4.5 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 3.4, 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 keyword 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.
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.

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

Figure 6: Top k algorithm using structure index

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.

5. INSTANCE OPTIMALITY WITH 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 5.1 to the case when the relevance query is a bag of path expressions.

The evaluation of a simple keyword 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 6. We modify the idea of extent chaining introduced in Section 3.4 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 2-5 initialize the indexidList appropriately depending on whether *sep* is / or //.
- The terminating condition in Step 8 is similar to the one in the procedure compute_top_k.
- The evaluation of currDocResults in Step 12 (for a single document) can be performed using intra-document extent chaining described in Section 3.4.
- In Step 9, 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 4), 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 modify the wild guess definition to obtain what we call a *strict wild guess*, by excluding 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 4.4. In particular, the index evaluation cost is not counted for the purpose of this discussion.

THEOREM 1.: *Let q be a simple keyword path expression query. Let $\mathcal{D}$ be the class of all databases such that q is covered by structure index I. 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 strict wild guesses. Then, compute_top_k_with _sindex is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

5.1 Extension to Bag of Simple Keyword Path Expressions

We now extend the above algorithm to the case when the query is a bag of simple keyword 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 relevance function for Q (which merges the relevances of the p_i and takes a product with the proximity function ρ). This merge is similar to the merge algorithm in Figure 5. The algorithm is omitted for lack of space. Since the merging function is monotonic and ρ lies in the range $[0, 1]$, this algorithm can easily be shown to be correct for all well-behaved relevance functions. This algorithm naturally extends when Q has more than two simple path expressions.

We show that for the special case when the relevance function is not proximity-sensitive, this algorithm is instance optimal for an interesting class of bag queries, over the class of algorithms that do not make strict wild guesses, as defined above. A bag B of simple path expressions is defined to be disjoint if the trailing terms of

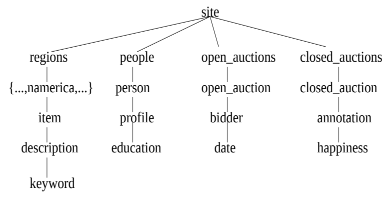

Figure 7: XMark Schema

no two simple path expressions in B are the same. For example, the bag {book//“XML”,author/“Abiteboul”} is disjoint while the bag {book//“XML”,article//“XML”} is not. The optimality of the algorithm is open for proximity-sensitive relevance functions.

Theorem 2.: *Let $Q = \{q_1, q_2, \ldots, q_l\}$ be a bag of simple keyword path expression queries. Let $\mathcal{D}$ be the class of all databases such that each q_i is covered by structure index I. We have the following.*

1. *For any database $D \in \mathcal{D}$, compute_top_k_bag correctly returns the top k documents over any given well-behaved relevance function.*
2. *Suppose that Q is a disjoint bag and that the relevance of a document is computed using a function that is well-behaved but not proximity-sensitive. 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 strict wild guesses. Then, compute_top_k_bag is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

6. EXPERIMENTAL RESULTS

We have implemented the above algorithms as part of a native XML database system [43]. We present the results of an experimental study that yields 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.

6.1 Evaluation of Branching Path Queries

We use the XMark XML-benchmark data [42] for this set of experiments. This data models an auction site. The element relationships relevant to this paper are shown in Figure 7. The tag names are self-explanatory. The data size is 100MB. The structure index we use is the 1-Index [32]. 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, 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 (we discuss the last column in this table later).

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

- The benefit of using a structure index in conjunction with inverted lists is considerable with speedups of as high as about 43 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.

In the literature, several techniques for inverted list joins have been proposed [8, 11, 25, 28, 37, 45]. While our technique reduces the number of joins irrespective of which of these algorithms is used to join inverted lists (i.e., as IVL), the actual speedup obtained as a result depends on the specific algorithm. The inverted list join algorithms can be broadly classified as follows: merge-based algorithms [28, 45], stack-based algorithms [8, 37] and extensions to these using secondary indexes [8, 11, 21, 25]. In the native XML database system we use, the inverted list join algorithm is a merge-based algorithm that uses auxiliary B-Tree indexes to skip parts of the inverted lists.

In order to study the impact of our technique on an alternative inverted list join algorithm, we consider the holistic twig join algorithm [8]. This algorithm uses a stack-based approach to join all inverted lists in one operator and also uses a data structure known as the XB-Tree to skip parts of the inverted lists. We study the speedup obtained through our technique by modifying the source code to incorporate index ids (without extent chaining). The speedups obtained on the XMark queries are shown in the final column of Table 1. The results obtained are consistent with what we get from our native XML database system. The fact that we obtain such speedups with two completely different implementations of inverted list joins shows that our techniques are widely applicable.

6.1.1 Effect of Extent Chaining

The above discussion shows the benefits obtained through our technique. This benefit is obtained by saving joins and using extent chaining to skip parts of an inverted list. Using an extent chain to access a list can be expensive compared to a linear scan, especially if the number of list entries matching an extent is high. In a separate set of experiments, we compare the performance of extent chaining and linear scan by varying the query selectivity.

Our methodology is the following. We create a document with the schema shown in Figure 9. That is, this document has a root element with a list of A children, each of which has a list of a fixed number, *fanout*, of A children. We use the schema in Figure 9 also as a path index to define indexids. The *id* of each index node is shown beside the node Figure 9.

The query we are interested in is finding all children of the root by scanning the inverted list for A. We vary *fanout* keeping the total number of entries in the inverted list for A with indexid 3 fixed at 250000. In the process, we compare the execution times for the following algorithms:

1. Scan the whole A list checking for each entry whether its *indexid* is 2.
2. Follow the extent chain for *indexid* 2.

Query in English	Path expression	Speedup with Native XML DBMS	Speedup with Holistic Twig Join
Find occurrences of “attires” under item descriptions	//item/description//keyword/“attires”	43.3	40.45
Find open auctions that had a bid in 1999	//open_auction[/bidder/date/“1999”]	6.85	2.89
Find the persons who attended Graduate school	//person[/profile/education/“Graduate”]	5.06	1.82
Find closed auctions where the happiness levelwas 10	//closed_auction[/annotation/happiness/“10”]	3.12	4.08

Table 1: Speedups Using Structure Index

(No. of A entries with indexid 2,*fanout*)	Time in sec. for seq. scan	Time in sec. for extent chain	Time in sec. hybrid scan
(25000,2)	1.32	4.36	1.69
(1000,50)	0.19	0.27	0.24
(500,100)	0.16	0.19	0.22
(250,200)	0.15	0.15	0.23
(100,500)	0.14	0.09	0.16
(50,1000)	0.14	0.05	0.1

Figure 8: Studying the overhead of extent chaining

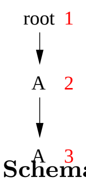

Figure 9: Schema for extent chaining experiments

The result is reported in Figure 8 (we explain the *hybrid scan* column later).

We observe the following from these numbers:

1. Even though sequential scan performs the same amount of I/O in each experiment, the time taken reduces as the number of entries copied over to the output decreases, showing the effect of this copy overhead.
2. In the worst-case, using the extent chain is 3.3 times worse than a sequential scan. The fanout value at which extent chaining begins to be beneficial is 200, which is roughly half a page of list entries. We note here that the distribution we have picked is the *worst-case* distribution for measuring the overhead of extent chaining. This is so since for a given selectivity, the maximum number of I/Os performed by extent chaining is when the results are uniformly spread out.
3. The previous observation regarding extent chaining is also applicable when parts of a list are skipped using a secondary index when performing an inverted list join.

In this context, in addition to the linear scan and extent chain scan approaches, we also consider a third *hybrid scan* approach: perform a linear scan, but if we find half a page of contiguous inverted list entries not matching the selected extents, follow the extent chain. The benefit of this approach is greatest when all entries matching the selected extents in an inverted list are clustered together. This approach then performs a scan over the matching portion and once it overshoots the matching portion by half a page, it uses the extent chain to skip the rest of the list. We implement the hybrid scan algorithm in our native XML database system and verify this behavior using an experiment on the XMark dataset. We run the query //africa/item. All item elements under africa are clustered together. Using the hybrid scan is 15.9 times faster than a linear scan.

On the other hand, the worst case for the hybrid scan is when the entries in an inverted list matching the selected extents are spread uniformly apart. Here, the number of unmatched entries examined per matched entry is highest. We test the performance of the hybrid scan algorithm over the data sets used for comparing extent chaining and linear scan. The results are reported in the final column of Figure 8. We observe that the performance of the hybrid scan algorithm is always close to that of the sequential scan, the worst-case overhead being 20%. Again, for a given selectivity, the overhead of the hybrid algorithm is worst when the results are uniformly spread out.

Based on the above observations, we conclude that the best strategy for an optimizer is the following: if the selectivity of the result indexids is below a certain threshold, then it must use the extent chain, otherwise it must use the hybrid scan algorithm. The worst case overhead of this algorithm is 20% more than the cost of a linear scan, while the best case benefits can be significant.

6.2 Relevance Queries

We have implemented the compute_top_k_with_sindex algorithm shown in Figure 6. 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 [5]. We pick a different data set for this study since we want a data set with multiple files to make ranked queries meaningful. 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,

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

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.

7. 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 input query is evaluated by traversing the data graph [21, 31] or some *compressed* representation [4, 9, 39]. The other involves information-retrieval style processing using inverted lists [8, 11, 25, 28, 37, 41, 45]. Methods have been proposed to optimize queries in the presence of both these alternatives [21, 24, 31]. In this framework, structure indexes such as the ones proposed in [18, 27, 32] have primarily been used as a substitute for graph traversal [31]. This paper proposes and evaluates an approach that merges structure indexes and inverted lists to evaluate arbitrary branching path expressions. We note that our techniques apply irrespective of which specific structure index and inverted list join algorithm is used. The technique we propose is similar to the algorithm proposed in [44] using path ids in the context of evaluating branching path expression queries over XML data stored in an RDBMS. That algorithm is correct for nonrecursive data sets — it turns out that it does not give the correct result when the input XML data has an ancestor and descendant element with the same tag name. Since path ids are a special form of structure indexes [32], and their technique is based on containment joins, the solution presented in this paper can be easily applied to their context.

In recent independent work in [40], the ViST index structure is proposed where structure and value are combined into a single index to evaluate path expression queries with structure and keyword components. The idea is to encode XML documents and queries as sequences and evaluate queries by finding subsequence matches, thus eliminating joins. First of all, the evaluation strategy using ViST involves a top-down traversal of a suffix tree which is unlikely to scale to single large documents. Hence, the ViST data structure works best for databases where there is a large number of small XML files. As anecdotal evidence, the authors show their results over the XMark database [42] by splitting a single document into smaller documents, each containing about 30 elements. However, a limitation of this approach is that queries that span the various fragments cannot be answered. In particular, deciding upon a method of partitioning a single document is not trivial. Our approach, on the other hand, handles large single documents. Secondly, as discussed in [40], in the presence of siblings with the same tag in the DTD, a branching path expression may need to be rewritten into multiple sequences (exponential in the worst case) for subsequence matching. This problem is more general in the absence of a DTD [20], where a query such as A[//B1][//B2]...[//Bk] where each B_i is different would be rewritten into $k!$ sequences. On the other hand, our algorithm performs atmost one join per tag and keyword occurrence in the query.

7.1 Ranked Search

Several proposals have been made for ranked search over a corpus of document databases combining keyword and structure components [22, 35]. Recently, in [2, 3, 15, 36, 38], query languages that integrate information retrieval related features such as ranking and relevance-oriented search into XML queries have been proposed. Techniques to evaluate these ranked queries are also proposed in [2, 3, 36, 38]. A survey of commercial XML search engines is available in [29]. In [33], the problem of ranking SGML documents using term occurrences is considered. As mentioned in Section 1, this paper considers a subclass of CAS queries and focuses on algorithmic issues in combining structure indexes with inverted lists to efficiently push down top k computation. To the best of our knowledge, none of these previous techniques uses structure indexes of the kind we describe in our paper to save joins and push down top k computation. Several previous projects have dealt with supporting ranked keyword search, like [1, 23] over structured databases, [6, 13, 17] over graph-structured data, [7, 14] over web sources and [19] over XML data. Our technique can be used to support a query language that extends keyword queries with a powerful additional search criterion, namely a path expression.

8. 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 experiments on a 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 to proximity-sensitive ranking functions. When the ranking function is well-behaved and proximity-insensitive, our algorithm is instance-optimal. 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, the problem of running structured queries over hyper-linked XML documents remains open.

9. REFERENCES

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

[2] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an XML database. In *SIGMOD*, 2003.

[3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, 2002.

[4] A. Arion et al. XQueC: Pushing queries to compressed XML data. In *VLDB*, 2003.

[5] X. A. A. at NASA. XML astronomy archive at NASA. `http://xml.gsfc.nasa.gov/archive`.

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

[7] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

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

[9] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, 2003.

[10] 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.

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

[12] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.

[13] S. Dar et al. DTL's Dataspot: database exploration using plain language. In *VLDB*, 1998.

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

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

[16] N. Fuhr, M. Lalmas, and S. Malik. INEX: initiative for evaluation of XML retrieval. `http://inex.is.informatik.uni-duisburg.de:2003`.

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

[18] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[19] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[20] Personal communication with Haixun Wang.

[21] A. Halverson et al. Mixed mode XML query processing. In *VLDB*, 2003.

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

[23] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

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

[25] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, 2003.

[26] J.Min, C.Chung, and K.Shim. APEX: An adaptive path index for xml data. In *SIGMOD*, 2002.

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

[28] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, 2001.

[29] R. Luk et al. A survery of search engines for XML documents. In *SIGIR Workshop on XML and IR*, 2000.

[30] Y. Mass, M. Mandelbrod, E. Amitay, Y. Maarek, and A. Soffer. JuruXML: An XML retrieval system. In *INEX Workshop*, 2002.

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

[32] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[33] S. Myaeng et al. A flexible model for retrieval of SGML documents. In *SIGIR*, 1998.

[34] J. Naughton et al. The Niagara internet query system. In *IEEE Data Engineering Bulletin*, 2000.

[35] G. Navarro and R. Baeza-Yates. Proximal nodes: A model to query document databases by content and structure. *ACM TOIS*, 1997.

[36] T. Schlieder and H. Meuss. Result ranking for structured queries against XML documents. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, 2000.

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

[38] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT*, 2002.

[39] P. Tolani and J. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.

[40] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, 2003.

[41] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *ICDE*, 2003.

[42] XMark: The XML benchmark project. `http://monetdb.cwi.nl/xml/index.html`.

[43] XXX. Native XML database system.

[44] M. Yoshikawa et al. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM TOIT*, 2001.

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