This is the **submitted version** of the paper that is to appear in SIGMOD 2003. The final camera-ready version will be available in mid-March 2003.

XRANK: Ranked Keyword Search over XML Documents

Lin Guo Feng Shao Chavdar Botev Jayavel Shanmugasundaram

Department of Computer Science
Cornell University
{guolin, fshao, cbotev, jai}@cs.cornell.edu

ABSTRACT

We consider the problem of efficiently producing ranked results for keyword search queries over hyperlinked XML documents. Evaluating keyword search queries over hierarchical XML documents, as opposed to (conceptually) flat HTML documents, introduces many new challenges. First, XML keyword search queries do not always return entire documents, but can return deeply nested XML elements that contain the desired keywords. Second, the nested structure of XML implies that the notion of ranking is no longer at the granularity of a document, but at the granularity of an XML element. Finally, the notion of keyword proximity is more complex in the hierarchical XML data model. In this paper, we present the XRANK system that is designed to handle these novel features of XML keyword search. Our experimental results show that XRANK offers both space and performance benefits when compared with existing approaches. An interesting feature of XRANK is that it naturally generalizes a hyperlink based HTML search engine such as Google. XRANK can thus be used to query a mix of HTML and XML documents.

1. INTRODUCTION

Keyword search querying has emerged as one of the most effective paradigms for information discovery, especially over HTML documents in the World Wide Web. One of the key advantages of keyword search is simplicity – users do not have to learn a complex query language, and can issue queries without any prior knowledge about the structure of the underlying data. Since the keyword search query interface is very flexible, queries may not always be very focused and can return a potentially large number of query results, especially when issued over large document collections. Consequently, an important requirement for keyword search is to rank the query results so that the most relevant results appear first.

Despite the success of HTML-based keyword search engines, certain limitations of the HTML data model make such systems ineffective in many domains. These limitations stem from the fact that HTML is a presentation language and hence cannot capture much semantics. The XML data model addresses this limitation by allowing for *extensible element tags*, which can be arbitrarily nested to capture additional semantics. As an illustration, consider the repository of conference and workshop proceedings shown in Figure 1. Each conference/workshop has the full-text of all its papers. In addition, information such as titles, references, sections and sub-sections are explicitly captured using nested, application-specific XML tags, which is not possible using HTML.

Given the nested, extensible element tags supported by XML, it is natural to exploit this information for querying. One approach is to use sophisticated query languages such as XQuery [34] to query XML documents. While this approach can be very effective in some cases, a downside is that users have to learn a complex query language and understand the schema of underlying XML. An alternative approach, and the one we consider in this paper, is to retain the simple keyword search query interface, but exploit XML's tagged and nested structure *during query processing*.

Keyword searching over XML introduces many new challenges. First, the result of the keyword search query is not always the entire document, but can be a deeply nested XML element. As an illustration, consider the keyword search query "XQL language" over the document shown in Figure 1. The keywords occur in a sub-section (line 15) and clearly, it will be good to return the XML element corresponding to the sub-section rather than returning the entire workshop proceedings (as would be done in a standard HTML search). In general, XML keyword search results can be arbitrarily nested elements, and returning the "deepest" node containing the keywords usually gives more context information (see also [16][29]).

Second, XML and HTML keyword search queries differ in how query results are ranked. HTML search engines such as Google usually rank documents based (partly) on their hyperlinked structure [6][23]. Since XML keyword search queries can return nested elements, ranking has to be done at the granularity of XML elements, as opposed to entire XML documents. For example, different papers in the XML document in Figure 1 can have different rankings depending on the underlying hyperlinked structure. Computing rankings at the granularity of elements is complicated by the fact that the semantics of containment links (relating parent and child elements) is very different from that of hyperlinks (such as IDREFs and XLinks [34]). Consequently, techniques for computing rankings solely based on hyperlinks [6][23] are not directly applicable for nested XML elements.

Finally, the notion of proximity among keywords is more complex for XML. In HTML, proximity among keywords translates directly to the distance between keywords in a document. However, for XML, the distance between keywords is just one measure of proximity; the other measure of proximity is the distance between keywords and the result XML element. As an illustration, consider the keyword search query "Soffer XQL". Although the keywords "Soffer" (line 3) and "XQL" (line 6) do not occur very far apart, the XML element containing both the keywords (the <workshop> element) is not a direct parent (<subsection>) of either keyword, and is thus not very proximal to either keyword. Intuitively, for XML, we need to consider a two-dimensional proximity metric involving both the keyword distance (i.e., width in the XML tree) and ancestor distance (i.e., height in the XML tree).

The above novel aspects of XML keyword search have interesting implications for the design of a search engine. In this paper, we

```
01. <workshop date="28 July 2000">
02.    <title> XML and IR: A SIGIR 2000 Workshop </title>
03.    <editors> David Carmel, Yoelle Maarek, Aya Soffer </editors>
04.    <proceedings>
05.      <paper id="1">
06.         <title> XQL and Proximal Nodes </title>
07.         <author> Ricardo Baeza-Yates </author>
08.         <author> Gonzalo Navarro </author>
09.         <body>
10.            <section name="Introduction">
11.               Searching on structured text is more important …
12.            </section>
13.            <section name="Implementing XQL Operations">
14.               <subsection name="Path Expressions">
15.                  At first sight, the XQL query language looks …
16.               </subsection>
17.               …
18.            </section>
19.            <cite ref="2">Querying XML in Xyleme</cite>
20.            <cite xlink="../paper/xmlql/">A Query … </cite>
21.         </body>
22.      </paper>
23.      <paper id="2">
24.         <title> Querying XML in Xyleme </title>
25.          …
26.      </paper>
27.    </proceedings>
28. </workshop>
```

Figure 1: An Example XML Document

describe the architecture, implementation and evaluation of the XRANK system built to address the above requirements for effective XML keyword search. Specifically, the contributions of the paper are: (a) the problem definition and system architecture for ranked keyword search over hierarchical and hyperlinked XML documents (Section 2), (b) an algorithm for computing the ranking of XML elements that takes into account both hyperlink and containment edges (Section 3), (c) new inverted list index structures and associated query processing algorithms for evaluating XML keyword search queries (Section 4), and (d) an experimental evaluation of XRANK and a comparison with alternative approaches (Section 5).

One of our design goals was to naturally generalize a hyperlink based HTML search engine such as Google [6]. XRANK is thus designed such that when the number of levels in the XML hierarchy is two (i.e., a document containing keywords), our system behaves just like a HTML search engine. Thus, XRANK allows for a graceful transition from HTML documents to XML documents (such as in the World Wide Web and Corporate Intranets) because it can handle both classes of documents using the same framework.

2. DATA MODEL & QUERY SEMANTICS

In this section, we first briefly describe the XML data model and then define the semantics for ranked keyword search queries over hyperlinked XML documents.

2.1 XML Data Model

The eXtensible Markup Language (XML) is a hierarchical format for data representation and exchange. An XML document consists of nested XML elements starting with the root element. Each element can have attributes and values, in addition to nested sub-elements. Figure 1 shows an example XML document representing the proceedings of a conference. The <workshop> element is the root element, and it has <title>, <editors> and <proceedings> sub-elements nested under it. The <conference> element also has the date attribute whose value is "*28 July 2000*". For ease of exposition, we treat attributes as though they are sub-elements.

In addition to the hierarchical element structure, XML also supports intra-document and inter-document references. Intra-document references are represented using IDREFs [34]. An example of an IDREF is shown in Figure 1, line 19, where one of the papers in the proceedings references another paper in the same proceedings. Inter-document references are represented using XLink [34]. An example is shown in Figure 1, line 20, where a paper in the proceedings references another paper in a different conference. We refer to both IDREFs and XLinks as hyperlinks.

Based on the above discussion, we can define a collection of hyperlinked XML documents to be a directed graph $G = (V, CE, HE)$, where V is the set of vertices that consists of XML elements and values. CE is the set of containment edges relating vertices; specifically, the edge $(u, v) \in CE$ iff v is a value/nested sub-element of u. HE is the set of hyperlink edges relating vertices; and the edge $(u, v) \in HE$ iff u contains a hyperlink reference to v. Vertex u is an *ancestor* of a vertex v if there is a sequence of containment edges that lead from u to v. The predicate $contains(v, k)$ is true if the vertex v (directly or indirectly) contains the keyword k.

2.2 Keyword Query Results

We now define the results of keyword search queries over XML documents (we defer the notion of ranking the results until the next section). We support two different semantics for keyword search queries. Under *conjunctive* keyword query semantics, elements that contain *all* of the query keywords are returned. Under *disjunctive* keyword query semantics, elements that contain *at least one* of the query keywords are returned. In the interest of space, we focus on conjunctive keyword query semantics in this paper.

Consider a keyword search query consisting of n keywords $k_1,\ldots, k_n$. Let $N = \{1, \ldots, n\}$ and $R_0 = \{v \mid v \in V \wedge \forall i \in N\ contains(v, k_i)\}$. The query result is the union of the following two disjoint sets:

1) $\{ v \mid v \in R_0 \wedge \forall c\ (v, c) \in CE \Rightarrow c \notin R_0 \}$

2) $\{ v \mid \exists c\ ((v, c) \in CE \wedge c \in R_0)$
$\quad \wedge\ \exists d,i\ ((v, d) \in CE \wedge d \notin R_0\ \wedge i \in N \wedge contains(d, k_i))\}$

Intuitively, (1) is the set of nodes that contain all of the query keywords, such that none of its children (and hence, descendants) contain all of the query keywords. (1) ensures that only the most specific results are returned to the user. For example, in Figure 1, the query 'XQL language' will return the corresponding <subsection> (lines 14-16). However, the <section> and <body> ancestors will not be returned because they have a descendant (<subsection>) that contains all of the query keywords.

The definition of (2) is subtler, and it ensures that every query keyword instance is represented in the query result. Intuitively, (2) is the set of nodes that have at least one child that contains all the query keywords, and at least one other child that contains some other instances of the query keywords. This condition is best explained with an example. Consider again the query 'XQL language'. If we just considered the set (1), the <paper> element (line 5) would not appear in the result because one of its descendants (<subsection>) appears in the result. However, the keyword 'XQL' appears in the <title> sub-element (line 6) of <paper>, and this keyword is not represented in any of the result elements in (1). Therefore, we include the <paper> element in the result set because it is the least ancestor of the keyword 'XQL' (in <title>) that also contains the other query keywords.

Note that we only consider containment edges when defining the results of a keyword search query. This is similar to many HTML document keyword search paradigms, where only the documents that contain the desired keywords are returned. Hyperlinks are mainly used to compute the ranking of the query results. The only exception is anchor text, which we assume is contained in the element pointed to by the corresponding hyperlink edge; this is similar to the strategy used in Google [6].

While returning nested XML elements provides more context information, it also poses interesting user-interface challenges. As an illustration, consider the keyword search query 'XML workshop' issued over the document in Figure 1. A result for this query is the <title> element. However, the title element may be too specific for the user because it does not present any information about whether it is a title of a book, journal or workshop. One solution is to allow the user to navigate up to the ancestors of the query result to get more context information when desired. Another solution, originally proposed in the context of keyword searching graph databases [4][13], is to predefine a set of "answer nodes" AN. As an example of the latter approach, a domain expert can determine that only <workshop>, <section>, and <subsection> elements are in AN, and consequently, only these elements can be the result of a keyword search query. An interesting application of pre-defining a set of answer nodes is to query a mix of XML and HTML documents. All the HTML tags (that are used for presentation purposes) are excluded from the AN set, and hence, the result set contains only entire HTML documents.

XRANK supports both user navigation for context information and the ability to pre-define answer nodes. Note that pre-defining answer nodes for XML documents may require knowledge of the domain and underlying XML schema. If such knowledge is not available, all XML elements can be treated as potential answer nodes. For the rest of this paper, we assume that every element is an answer node.

2.3 Ranking Keyword Query Results

We now turn to the issue of ranking the results of keyword search queries over XML documents. We first described some desired properties of the ranking function before defining it more formally.

2.3.1 Ranking Function: Desired Properties

We believe that a ranking function for keyword search queries over a large collection of hyperlinked XML documents should have the following properties:

1) *Result specificity*: The ranking function should rank more specific results higher than less specific results. For example, in Figure 1, a <subsection> result (which means that all query keywords are in the same subsection) should be ranked higher than a <section> result (which means that the query keywords occur in different subsections). This is one dimension of result proximity.

2) *Keyword proximity:* The ranking function should take the proximity of the query keywords into account. This is the other dimension of result proximity. Note that a result can have high keyword proximity and low specificity, and vice-versa.

3) *Hyperlink Awareness*: The ranking function should use the hyperlinked structure of XML documents. For example, in Figure 1, widely referenced papers should be ranked higher.

While traditional information retrieval systems [28] and HTML search engines [6] take 2 and 3 into account, 1 is specific to XML keyword search. Some recent work on searching graph databases [4][13] considers a variant of 1 and some part of 3, but does not consider 2. Our goal in this section is to formalize the notion of ranking for XML elements by taking all of the above factors into account. Further, we would like the generalization to also work for HTML documents (where 1 is not of concern).

2.3.2 Ranking Function: Definition

We now define the ranking function for keyword search queries over XML documents. For the purposes of this section, we will just assume that *ElemRank(v)* is the objective importance of an XML element v computed using the underlying hyperlinked structure. Conceptually, *ElemRank* is similar to Google's *PageRank* [6], except that *ElemRank* is defined at the granularity of an element and takes the nested structure of XML into account. More details on *ElemRank* are presented in Section 3.

Consider a keyword search query $Q = (k_1, k_2, \ldots, k_n)$ and the corresponding result set R. Now consider a result element $v_1 \in R$. We first define the ranking of v_1 with respect to one query keyword k_i, $r(v_1, k_i)$, before defining the overall rank, $rank(v_1, Q)$.

2.3.2.1 Ranking with respect to one keyword

By the definition of R, we know that $contains(v_1, k_i)$ is true for every k_i, Hence, there is a sequence of containment edges of the form $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ such that v_n directly contains k_i. We define:

$$\mathrm{r}(v_1, k_i) = ElemRank(v_n) \times decay^{n-1}$$

Intuitively, the rank of v_1 with respect to a keyword k_i is $ElemRank(v_n)$, where v_n directly contains k_i, scaled appropriately to account for the specificity of the result. When the result element v_1 directly contains the keyword (i.e., $v_1 = v_n$), the rank is just the *ElemRank* of the result element. When the result element indirectly contains the keyword (i.e., $v_1 \neq v_n$), the rank is scaled down by a factor *decay* for each level. *decay* is a parameter that can be set to a value in the range 0 to 1.

The astute reader may have noticed that $r(v_1, k_i)$ does not depend on the *ElemRank* of the result node v_1, except when $v_1 = v_n$. We chose to have $r(v_1, k_i)$ depend on the *ElemRank* of v_n rather than the *ElemRank* of v_1 for the following two reasons. First, by scaling down the same quantity - $ElemRank(v_n)$ - we ensure that less specific results indeed get lower ranks. Second, as we shall see in Section 3, the $ElemRank(v_n)$ is in fact related to $ElemRank(v_1)$ due to certain properties of containment edges.

In the above formula, we have implicitly assumed that the query keyword k_i occurs only once in the result element. In case there are multiple (say, m) occurrences of k_i, we first compute the rank for each occurrence using the above formula. Let the computed ranks be $r_1, r_2, \ldots, r_m$. The combined rank is:

$$\hat{r}(v_1, k_i) = f(r_1, r_2, \ldots, r_m)$$

Here f is some aggregation function. In most of our experiments, f = ***max***, but other choices (such as $f = sum$) are also supported.

2.3.2.2 Overall Ranking

The overall ranking of a result element v_1 for query $Q = (k_1, k_2, \ldots, k_n)$ is computed as follows.

$$R(v_1, Q) = \left(\sum_{1 \le i \le n} \hat{r}(v_1, k_i) \right) \times p(v_1, k_1, k_2, \ldots, k_n)$$

The overall ranking is the sum of the ranks with respect to each query keyword, multiplied by a measure of keyword proximity $p(v_1, k_1, k_2, \ldots, k_n)$. We currently set the keyword proximity to be inversely proportional to the minimum window size that contains all the query keywords in v_1 (the maximum value of keyword proximity is 1 and minimum value is 0.2). Clearly, other combination functions to produce the overall rank are also possible. XRANK is general enough to handle any combination function so long as the *first factor* in the above formula is monotone with respect to individual keyword ranks (the reason for the monotone restriction will be clarified in Section 4.3). In some cases, users may also wish to assign different weights to different keywords, in which case the individual keyword ranks are weighted accordingly.

2.4 XRANK System Architecture

The components of the XRANK system are shown in Figure 2. The *ElemRank* Computation module computes the *ElemRank*s of XML elements. The *ElemRank*s are then combined with ancestor information to generate an index structure called HDIL (Hybrid Dewey Inverted List). The Query Evaluator module evaluates queries using HDIL, and returns ranked results. In subsequent sections, we describe these components in more detail.

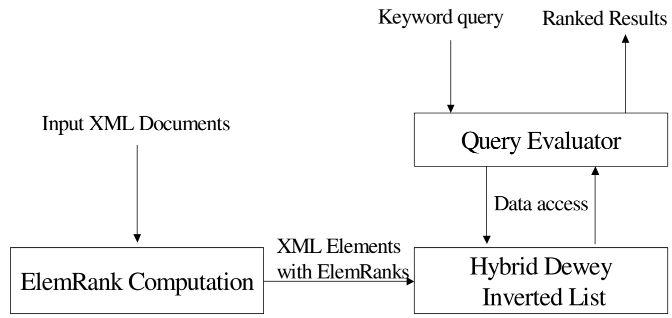

Figure 2: XRANK Architecture

3. COMPUTING *ElemRank*s

We now consider the problem of computing *ElemRank*s for XML elements. As mentioned earlier, *ElemRank* is a measure of the objective importance of an XML element, amd is computed based on the hyperlinked structure of XML documents. *ElemRank* is similar to Google's *PageRank*, but is computed at the granularity of an element and takes the nested structure of XML into account. Note that we need to compute ranks at the granularity of elements because different elements in the same XML document can have very different ranks. For example, in Figure 1, the importance of different <paper> elements can vary widely.

We now develop our *ElemRank* algorithm as a series of refinements to the *PageRank* algorithm [6] (these also work for query-dependent algorithms like HITS [23]). The refinements retain the original ranking semantics for HTML documents, and also help identify the main differences between computing ranks for HTML and XML documents. We also evaluate the computational cost of our algorithm on real and synthetic datasets.

3.1 Algorithm for Computing *ElemRank*

The algorithm for computing *PageRank*s [9] of HTML documents works by repeated applications of the following formula (N_d is the total number of documents, and $N_h(v)$ is the number of out-going hyperlinks from document v):

$$p(v) = \frac{1-d}{N_d} + d \times \sum_{(u,v) \in HE} \frac{p(u)}{N_h(u)}$$

As shown, the *PageRank* of a document v, $p(v)$, is the sum of two probabilities. The first is the probability $(1-d)/N_d$ of visiting v at random (d is a parameter of the algorithm, usually set to 0.85). The second is the probability of visiting v by navigating through other documents. In the second case, the probability is calculated as the sum of the normalized *PageRank*s of all documents that point to v, multiplied by d, the probability of navigation [6].

Let us now try to directly adapt this formula for use with XML documents by mapping each element to a document, and by mapping all edges (IDREF, XLink and containment edges) to hyperlink edges. One of the main problems with this adaptation is that hyperlinks are treated as directed edges, and the *PageRank* propagates along only one direction[1] [6]. This unidirectional *PageRank* propagation for HTML documents corresponds to the intuition that if an important page *p1* points to a page *p2*, then *p2* is likely to be important. However, if *p1* points to an important page *p3*, that does not tell us anything about the importance of *p1* (consider relatively obscure HTML pages that point to Yahoo).

In the case of containment edges, however, there is a tighter coupling between the elements. As an illustration, consider the XML document in Figure 1. If a paper element has a high *ElemRank*, then it is natural that the sections of the paper also have high *ElemRank*s; this corresponds to *forward ElemRank* propagation along containment edges. In addition, if a workshop contains many papers that have high *ElemRank*s, then the workshop should also have a high *ElemRank*; this corresponds to *reverse ElemRank* propagation. More generally, containment implies a tighter relationship (the corresponding elements are present in the same document) than hyperlinks, and hence argues for a ***bi-directional transfer of ElemRanks***.

A simple solution is to add reverse containment edges, as shown below. $e(v)$ is used to denote the *ElemRank* of an element v.

$$e(v) = \frac{1-d}{N_e} + d \times \sum_{(u,v) \in E} \frac{e(u)}{\left(N_h(u) + N_c(u) + 1\right)}$$

[1] This is typical of most algorithms for hyperlinked HTML documents. For example, the HITS algorithm [23] propagates all authority values along the same direction (only a different measure, hub values, is propagated along the reverse direction).

N_e is the total number of XML elements, $N_c(u)$ is the number of children contained by u, and $E = HE \cup CE \cup CE^{-1}$, where CE^{-1} is the set of reverse containment edges.

While the above formula supports bi-directional transfer of *ElemRank*s along containment edges, it still has a shortcoming - it does not distinguish between containment and hyperlink edges when computing *ElemRank*s. As an illustration, consider a paper that has few sections and many references. As per the above formula, the *ElemRank* of the paper are uniformly distributed among all the sections and references. Thus, the larger the number of references in a paper, the less important each section of the paper is likely to be, which is not very intuitive. In general, the problem is hyper-links and containment edges are treated similarly, even though these two factors are usually independent. This argues for ***discrimination between containment and hyperlink edges*** when computing *ElemRank*s, as shown below.

$$e(v) = \frac{1 - d_1 - d_2}{N_e} + d_1 \sum_{(u,v) \in HE} \frac{e(u)}{N_h(u)} + d_2 \sum_{(u,v) \in CE \cup CE^{-1}} \frac{e(u)}{N_c(u) + 1}$$

d_1 and d_2 are the probabilities of navigating through hyperlinks and containment links, respectively.

The above formula still has a problem – it weights forward and reverse containment relationships similarly. To see why this is a problem, consider again the example in Figure 1. If a paper has many sections, then we would like the *ElemRank* of each section to be a fraction of the *ElemRank* of the whole paper. More generally, *ElemRank*s of sub-elements should be inversely proportional to the number of sibling sub-elements, as captured in the above formula. However, the *ElemRank* of a parent element should be directly proportional to the aggregate of the *ElemRank*s of its sub-elements. For instance, a workshop that contains many important papers should have a higher *ElemRank* than a workshop that contains only one important paper. This semantics of ***aggregate ElemRanks for reverse containment relationships*** is not captured above.

We now present our final formula that addresses the above issues. d_1, d_2, and d_3 are the probabilities of navigating through hyperlinks, forward containment edges, and reverse containment edges, respectively. $N_{de}(v)$ is the number of elements in the XML documents containing the element v.

$$e(v) = \frac{1 - d_1 - d_2 - d_3}{N_d \times N_{de}(v)} + d_1 \sum_{(u,v) \in HE} \frac{e(u)}{N_h(u)} + d_2 \sum_{(u,v) \in CE} \frac{e(u)}{N_c(u)} + d_3 \sum_{(u,v) \in CE^{-1}} e(u)$$

Note that we have also scaled down the first term (the probability of randomly visiting an element) by the number of elements in the document. This scaling ensures that *ElemRank* propagation along reverse containment edges is not biased towards large documents.

While we have motivated *ElemRank* using the example in Figure 1, it also has a more general interpretation in the context of random walks over XML graphs (this is a generalization of the random walk interpretation in [6]). Consider a random surfer over a hyperlinked XML graph. At each instant, the surfer visits an element e, and performs one of the following actions: (1) with probability $1\text{-}d_1\text{-}d_2\text{-}d_3$, he jumps to a random document, and then to a random element within the document, (2) with probability d_1, he follows a hyper-link from e, (3) with probability d_2, he follows a containment edge to one of e's children, and (4) with probability d_3, he goes to e's parent element. Given this model, $e(v)$ is exactly the probability of finding the random surfer in element v.

In most XML/HTML document collections, certain elements may not have hyperlinks, others may not have child elements, and some others (the document roots) may not have parent elements. In such cases, the probability of navigation ($d_1+d_2+d_3$) is proportionally split among the available alternatives. The proof of convergence for the *ElemRank* computation is similar to that described in [23], and is omitted in the interest of space.

3.2 Experimental Results

We ran the *ElemRank* computation algorithm on both real (DBLP) and synthetic (XMark [30]) datasets. The experiments were run using a 1.7GHz Pentium processor with 1GB of main memory and 30GB of disk space. We set the parameters $d_1 = 0.35$, $d_2 = 0.25$, $d_3 = 0.25$, and set the convergence threshold to 0.00005. The computation for the entire (124MB) DBLP dataset and 113MB XMark dataset converged within 10 and 5 minutes, respectively. This suggests that computing *ElemRank*s at the granularity of elements (as opposed to the granularity of a document) is feasible for reasonably large XML document collections. We have not tried to compute *ElemRank*s for document collections of the scale of the World Wide Web, mainly because the WWW does not contain such large XML collections (yet). However, we believe that the proposed algorithm will be applicable for large-scale XML repositories because the *ElemRank* computation is done offline, and does not affect keyword query evaluation time (see Figure 2).

In Section 5, we will present anecdotal evidence that *ElemRank*s computed using the above parameter settings, used with keyword proximity information, produces intuitive overall rankings. We have also varied the values of d_1, d_2, and d_3, and found that while it changes the relative weighting of hyperlinks and containment edges, it does not have a significant effect on algorithm convergence time.

4. EFFICIENTLY EVALUATING XML KEYWORD SEARCH QUERIES

We now turn to the main focus of this paper, which is efficiently producing ranked results for XML keyword search queries. This section is more general in scope than the previous section in that it does not depend on a particular method for computing XML element ranks. Although we shall use *ElemRank* to illustrate our techniques, they are applicable to other ways of ranking XML elements, such as those using text tf-idf measures [28][32]. We first present a naïve approach as a motivation for our techniques.

4.1 Naïve Approach

One main difference between XML and HTML keyword search is the granularity of the query results – XML keyword search returns elements while HTML keyword search returns entire documents. Thus, one way to do XML keyword search is to treat each element as a document, and use regular document-oriented keyword search methods. This approach, however, has the following problems.

1) Space overhead. Inverted list indices [28] are typically used to speed up the evaluation of keyword search queries. An inverted list contains for each keyword, the list of documents that contain the keyword. A naïve adaptation of inverted lists for XML elements would contain for each keyword, the list of elements that contain the keyword. This would result in a large space overhead because each inverted list would not only contain the XML element that

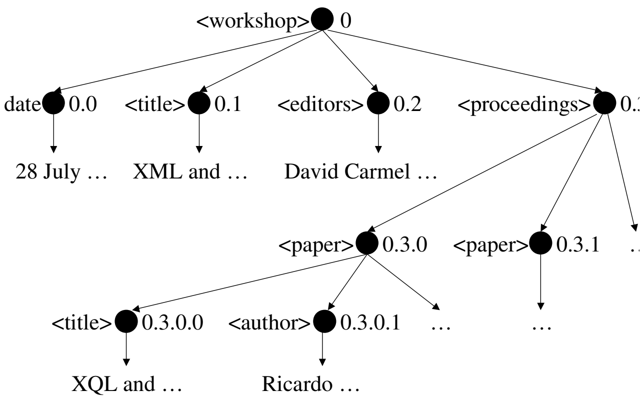

Figure 3: Dewey IDs

directly contains the keyword, but would also redundantly contain *all* of its ancestors (because they too contain the keyword).

2) Spurious query results. The naïve approach ignores ancestor-descendant relationships and treats all elements as though they are independent documents. Thus, if a sub-element appears in the query result, all of its ancestors will also appear in the query result (because if a sub-element contains the query keywords, all of its ancestors will also contain the query keywords). This will generate spurious query results, and will not correspond to our desired semantics for XML keyword search (see Section 2.2).

3) Inaccurate ranking of results. Existing approaches do not take result specificity into account when ranking results (Section 2.3.1).

We now present data structures and query-processing techniques that address the above limitations of the naïve approach.

4.2 Dewey Inverted List (DIL)

One of the drawbacks of the naïve approach is that it decouples the representation of ancestors and descendants. Consequently, it suffers from increased space overhead (because ancestor information is replicated) and spurious query results (because every ancestor of a query result is also returned). We now describe the Dewey encoding of element IDs, which jointly captures ancestor and descendant information.

Consider the tree representation of an XML document, where each element is assigned a number that represents its relative position among its siblings. The path vector of the numbers from the root to an element uniquely identifies the element, and can be used as the element ID. Figure 3 shows how Dewey elements IDs are generated for the XML document in Figure 1. An interesting feature of Dewey IDs is that the ID of an ancestor is a prefix of the ID of a descendant. Consequently, ancestor-descendant relationships are implicitly captured in the Dewey ID.

The idea of Dewey IDs is not new, and it has been used in the context of general knowledge classification, tree addressing [20], querying LDAP hierarchies [22] and ordered XML data [31]. Our focus, however, is to use Dewey IDs to support XML keyword search. As we shall see shortly, this new problem setting requires the development of novel algorithms.

4.2.1 DIL: Data Structure

Figure 4 shows the Dewey Inverted List (DIL) for the XML tree in Figure 3. The inverted list for a keyword *k* contains the Dewey IDs

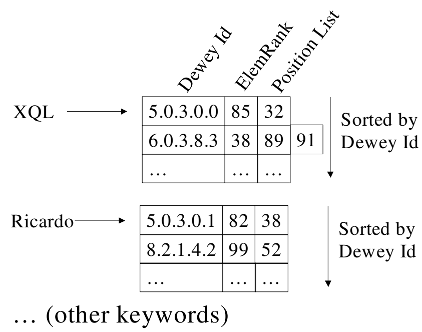

Figure 4: Dewey Inverted List

of all the XML elements that *directly* contain the keyword *k*. To handle multiple documents, the first component of each Dewey ID is the document ID. Associated with each Dewey ID entry in DIL is the *ElemRank* of the corresponding XML element, and the list of positions where the keyword *k* appears in that element (*posList*). The entries are sorted by the Dewey IDs. Since DIL only stores the IDs of elements that directly contain the keyword, its size is likely to be much smaller than the size of the naïve inverted list.

The observant reader might have noticed that even though DIL has a smaller number of entries, the size of each Dewey ID is larger. Fortunately, it turns out that the space overhead of Dewey IDs is more than offset by the space savings obtained by storing a smaller number of entries (we will present experimental results to validate this claim in Section 5). The relatively modest space overhead of Dewey IDs is attributable to the fact that each component of the Dewey ID is the *relative* position of an element with respect to its siblings. Consequently, a small number of bits are usually sufficient to encode each component of a Dewey id.

4.2.2 DIL: Query Processing

While DIL reduces space, it introduces new challenges for query processing. First, unlike traditional inverted list processing, one cannot simply do an equality merge-join of the query keyword inverted lists because the result IDs have to be *inferred* from the IDs of descendants. Second, spurious results must be suppressed. We now describe an algorithm that addresses these issues, and works in a *single pass* over the query keyword inverted lists.

The key idea is to merge the query keyword inverted lists, and *simultaneously* compute the longest common prefix of the Dewey IDs in the different lists. Since each prefix of a Dewey ID is the ID of an ancestor, computing the longest common prefix will automatically compute the ID of the deepest ancestor that contains the query keywords (this corresponds to computing sets (1) and (2) in Section 2.2). Since the inverted lists are sorted on the Dewey ID, all the common ancestors are clustered together, and this computation can be done in a single pass over the inverted lists.

The pseudo-code for the query processing algorithm is shown in Figure 5. The inputs to the algorithm are n query keywords (k1, …, kn), and the desired number of top-ranked query results (m). The algorithm works for n > 1, and the case where n = 1 is handled as a (simple) special case. The algorithm maintains two data structures, the result heap and the Dewey stack. The result heap keeps track of the top m results seen so far. The Dewey stack stores the ID, rank and position list of the current Dewey ID, and also keeps track of the longest common prefixes computed during the merge of the inverted lists.

```
01. procedure EvaluateQuery (k1, k2, …, kn, m) returns idList
02.   // k1 … kn are the query keywords,  m is the desired number of query results
03.   // invertedList[i] is the inverted list for keyword ki

04.   resultHeap = empty;   // Intialize the result heap of size m
05.   deweyStack = empty;  // Initialize the Dewey stack

06.   while (eof has not been reached on all inverted lists) {

07.       // Read the next entry from the inverted list having the smallest DeweyID
08.       find ilIndex such that the next entry of invertedList[ilIndex] is the smallest DeweyID
09.       currentEntry = invertedList[ilIndex].nextEntry;

10.       // Find the longest common prefix between deweyStack and currentEntry.deweyId
11.       find largest lcp such that deweyStack[i] = currentEntry.deweyId[i], 1 <= i <= lcp

12.       // Pop non-matching entries in the Dewey stack; add to result heap if appropriate
13.       while (deweyStack.size > lcp) {
14.           stackEntry = deweyStack.pop();
15.           if (stackEntry.potentialResult and stackEntry.posList non-empty for all keywords)
{
16.               compute overall rank using formula in Section 2.3.2.2
17.               if overall rank is among top m seen so far, add deweyStack ID to resultHeap
18.           }
19.       }

20.       // Update the rank and position lists of the longest common prefix entries
21.       for (all i such that 1 <= i <= lcp) {
22.           deweyStack[i].rank[ilIndex] = rank computed using formula in Section 2.3.2.1
23.           deweyStack[i].posList[ilIndex] += currentEntry.posList;
24.       }

25.       // Add non-matching components of currentEntry.deweyId to deweyStack
26.       for (all i such that mcp < i <= currDeweyIdLen) {
27.           stackEntry.rank[ilIndex] = rank computed using formula in Section 2.3.2.1;
28.           stackEntry.posList[ilIndex] = currentEntry.posList;
29.           deweyStack.push(deweyStackEntry);
30.       }

31.       // Set the longest common prefix entry to be a potential result
32.       deweyStack[lcp].potentialResult = true;
33.   } // End of looping over all inverted lists

34.   pop entries of deweyStack and add to result heap if appropriate (similar to lines 13-19)
35.   return ids in resultHeap;
```

Figure 5: DIL Query Processing Algorithm

The algorithm works by merging the inverted lists by the Dewey ID (lines 6-9), and computing the longest common prefix of the current entry and the previous entry stored in the Dewey stack (lines 10-11). It then pops all the Dewey stack components that are not part in the common prefix (lines 12-19), and if any of the popped components are potential query results, they are added to the result heap (lines 15-18). The current entry is then pushed onto the Dewey stack and the ranks and posLists are updated accordingly (lines 20-30). The longest common prefix is set to be a potential result (lines 31-32). The longest common prefix will be added to the output heap in a later loop when it is popped from the Dewey stack (lines 15-18).

Dewey	Rank[1]	Rank[2]	PosList[1]	PosList[2]	PotentialResult
0	85		32		1
0	77		32		0
3	68		32		0
0	61		32		0
5	56		32		0

(a)

Dewey	Rank[1]	Rank[2]	PosList[1]	PosList[2]	PotentialResult
1		82		38	0
0	77	74	32	38	1
3	68	66	32	38	0
0	61	60	32	38	0
5	56	54	32	38	0

(b)

Figure 6: States of Dewey Stack

We now walk through the algorithm using an example. Consider the DIL shown in Figure 4, and consider the keyword search query 'XQL Ricardo'. The algorithm first reads the entry with the smallest Dewey ID - *5.0.3.0.0*. Since the Dewey stack is initially empty, the longest common prefix is empty, and the Dewey ID components are simply pushed onto the stack, with the appropriate rank and posList fields (lines 25-32). The state of the stack is shown in Figure 6(a). Note that the ranks of the ancestors (prefixes) have been scaled down as per the ranking function (Section 2.3.2.1).

The algorithm then reads the next smallest entry, which is Dewey ID *5.0.3.0.1* in the 'Ricardo' inverted list. The longest common prefix (*5.0.3.0*) of the current entry and the Dewey stack is determined (lines 10-11), and non-matching entries are popped from the stack (12-19). The ranks and position lists of the longest common prefix components are updated (lines 20-24), and the longest common prefix is also marked as a potential result (line 31-32). The current state of the Dewey stack is shown in Figure 6(b). Note that ancestors of the longest common prefix are not marked as potential results, thereby eliminating spurious results.

The algorithm then reads the next smallest Dewey ID (*6.0.3.8.3*). Since the longest common prefix with the Dewey stack is empty, it pops the contents of the stack and adds the potential result (*5.0.3.0*) to the output heap. The algorithm then pushes *6.0.3.8.3* onto the stack and proceeds as before.

4.2.3 DIL: Correctness and Space/Time Complexity

It can be proved that the algorithm in Figure 5 correctly computes the top-m results as per the definition of query results and ranking described in Section 2.2. The actual proof is omitted in the interest of space. The space and time complexity of the algorithm are as follows. Let the query keywords be k_1, …, k_n, and let the corresponding number of entries in the inverted lists be L_1, …, L_n.

Further, let c be the maximum number of components in a Dewey ID (equivalently, c is the maximum XML document depth).

The time complexity of the algorithm is $O(c * (L_1 + \ldots + L_n))$, because each query keyword inverted list is scanned exactly once, and the cost of processing each inverted list entry using the Dewey stack is at most $O(c)$. The space complexity of the algorithm is $O(c + m)$, where c is for the Dewey stack and m is for the output heap.

4.3 Ranked Dewey Inverted List (RDIL)

Although DIL evaluates queries in a single pass over the query inverted lists, it suffers from a potential disadvantage. If inverted lists are long (due to common keywords or large document collections), even the cost of a single scan of the inverted lists can be expensive, especially if users want only the top few results. One solution is to order the inverted lists by the *ElemRank* instead of by the Dewey ID. In this way, higher ranked results are likely to appear first in the inverted lists, and query processing can usually be terminated without scanning all of the inverted lists. As a simple example, if a query contains just one keyword, only the first m inverted list entries have to be scanned to find the top m results.

Processing queries with multiple keywords is more challenging because one query keyword may occur in an element with a high *ElemRank* (which will appear at the beginning of its inverted list), while another keyword may appear in an element with low *ElemRank* (which will appear at the end of its inverted list). Many algorithms have been proposed for merging such ranked lists efficiently, but most of them (e.g., [3][9][27]) only work for disjunctive keyword queries. Recently, the Threshold Algorithm [14] has been proposed that works for conjunctive queries too. However, these approaches do not address the requirements of XML keyword search, including determining the most specific results, and handling non-monotone ranking functions. (Note that the ranking function in Section 2.3.2.2 is non-monotone with respect to *ElemRank* because we take result specificity and keyword proximity into account). We now describe RDIL that addresses the above issues.

4.3.1 RDIL: Data Structure

RDIL is similar to DIL, except that the inverted lists are ordered by *ElemRank* instead of Dewey ID. In addition, each inverted list has a B+-tree index on the Dewey ID field (the role of the B+-tree will be discussed shortly). Figure 7 illustrates the RDIL data structure. Although the figure shows a separate B+-tree for each inverted list, in reality this is too expensive in terms of space. This is because many inverted lists are very short, and wasting one whole disk page for indexing a short inverted list (of say, 200 elements) will blow up space requirements. Thus, in our implementation, we store multiple B+-trees (over short inverted lists) on the same disk page.

4.3.2 RDIL: Query Processing

The RDIL query processing algorithm is shown in Figure 8. The algorithm reads an entry from the query keyword inverted lists in a round-robin fashion (lines 8-10). Consider an entry retrieved from the inverted list of keyword k_i. The entry contains the Dewey ID d of a top-ranked element that contains at least one query keyword, which is k_i. However, to determine a query result, we need to determine the longest prefix of d that also contains the other query keywords. How can we determine such a prefix of d efficiently?

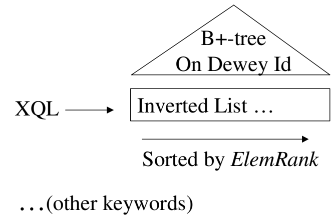

Figure 7: Ranked Dewey Inverted List

We now show how B+-trees can be used to determine the longest common prefix of d efficiently during query processing. Consider a query keyword k_j (<> k_i). To find the longest common prefix of d that also contains the keyword k_j, we just need to find the smallest Dewey ID, d_2, in the k_j inverted list that is larger than d. (Note that this operation can be easily supported in B+-trees because it is logically equivalent to starting a range scan at d, and reading the first entry in the range.) Either d_2 or its immediate predecessor in the B+-tree, d_3, shares the longest common prefix with d.

As an illustration, consider the keyword search query 'XQL Ricardo', and consider a top-ranked Dewey ID, *9.0.4.2.0*, that contains the keyword 'XQL'. Now, assume that the leaf nodes of the B+-tree for the 'Ricardo' inverted list have the Dewey IDs "*…, 8.2.1.4.2, 9.0.4.1.2, 9.0.5.6, 10.8.3, …*" (note that since the B+-tree is built on the Dewey IDs, the leaf nodes of the B+-tree are ordered by the Dewey ID even though the inverted list is ordered by *ElemRank*). To determine the longest common prefix of *9.0.4.2.0* that also contains the keyword 'XQL', we first determine the smallest Dewey ID in the B+-tree that is larger than 9.0.4.2.0, which in our example is *9.0.5.6*. Then either *9.0.5.6* or its predecessor in the B+-tree, *9.0.4.1.2*, shares the longest common prefix with *9.0.4.2.0*. In our example, this longest common prefix of *9.0.4.2.0* that also contains 'Ricardo' is *9.0.4*.

The RDIL algorithm thus determines the longest common prefix of a Dewey ID that contains all the query keywords by repeatedly probing the B+-tree for each query keyword (lines 11-15). Once the longest common prefix is determined, ranks and posLists are obtained using regular B+-tree lookups, and the overall rank is computed. The query result is then added to the output heap (lines 17-25). Note that the overall rank of the longest common prefix can be much less than the rank of an entry in the inverted list. This is because ranks decay when the results become less specific, i.e., when the longest common prefix is short (see Section 2.3.2.1).

Given that longest common prefix IDs can potentially have low overall ranks, how can we determine when we have the top m results so that we can stop scanning the inverted lists? In order to derive a stopping condition that still *guarantees* to output the top-m results, we build upon the provably optimal Threshold Algorithm (TA) [14]. TA computes a threshold at every point during the scan of the inverted lists. If there are at least m elements in the output heap that have an overall rank greater than the current threshold, the algorithm can stop scanning the lists. In our context, this threshold is the sum of the *ElemRank*s of the last processed element in each query keyword inverted list (lines 26-28).

It is important to note that while TA assumes a monotonic function for computing the overall rank from the individual keyword ranks, our overall rank computation is non-monotone because we take result specificity and keyword proximity into account (see Section 2.3.2). However, since the maximum values of decay and keyword proximity can be at most 1, we just use this maximum value when

```
procedure EvaluateQuery (k1, …, kn, m) returns idList
  // k1 … kn are the query keywords, and m is the desired number of query results
  // invertedList[i] corresponds to the inverted list for keyword ki
  // btree[i] corresponds to the B+-tree over the inverted list for keyword ki

  resultHeap = empty; // Initialize the result heap to any size greater than m
  done = false;
  while (!done and eof has not been reached on all inverted lists) {
      // choose the next keyword inverted list to read from in a round-robin fashion
      ilIndex = inverted list chosen in round-robin fashion (1 <= ilIndex <= n)
      currEntry = invertedList[ilIndex].nextEntry;

      // Find the longest common prefix that contains all query keywords
      lcp = currEntry.deweyID;
      for (all j such that 1 <= j < n) {
          probeIndex = (currIndex + j)%n;
          lcp = btree[probeIndex].getLongestCommonPrefix(lcp);
      }

      // If the longest common prefix is not already present in the result heap,
      // compute its rank and add to result heap
      if (!resultHeap.contains(lcp)) {
          for (all j such that 1 <= j <= n) {
              Get the rank and posList of lcp for keyword kj using btree[j]
          }
          compute overall rank using formula in Section 2.3.2.2;
          add (lcp, overall rank) to resultHeap;
      }

      // Compute current threshold and check whether the algorithm can terminate
      threshold = ∑1<= ji<= n (invertedList[i].currEntry.ElemRank);
      if (threshold < rank of top m elements in result heap) done = true;
  }
  return the top m elements from the resultHeap;
```

Figure 8: RDIL Query Processing Algorithm

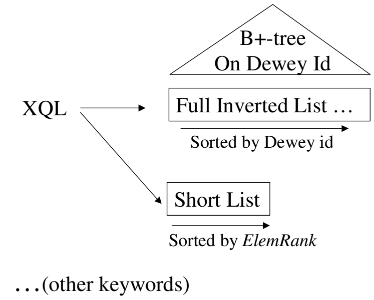

Figure 9: Hybrid Dewey Inverted List

computing the threshold. Since we only overestimate the threshold, the top *m* results are still guaranteed to be optimal.

4.3.3 RDIL: Correctness and Space/Time Complexity

It can be proved that the algorithm in Figure 8 correctly computes the top *m* ranked query results as per the definitions in Section 2.2. The proof is an extension of the proof of optimality of the Threshold Algorithm [14], and is omitted due to space constraints. Using the notion as in Section 4.2.3 the time complexity of the algorithm is $O(c * n * log_{1<=i<=n}(max(L_i)) * (L_1 + … + L_n))$, because in the worst case, each inverted list will have to be scanned completely $(L_1 + … + L_n)$, requiring *n* B+-tree probes for each entry ($n * log_{1<=i<=n}(max(L_i))$) that has Dewey ID of length *c*. Note that this is the worst case complexity, and RDIL can terminate much earlier. The space complexity of the algorithm is $O(c + m)$, which is the same as DIL.

4.4 Hybrid Dewey Inverted List (HDIL)

Even though RDIL is likely to perform well in many cases, there are certain cases where it is likely to perform much worse than DIL. For example, consider a query where the keywords are not very correlated, i.e., the individual query keywords occur relatively frequently in the document collection but rarely occur together in the same document. Since the number of results is small, RDIL has to scan most (or all) of the inverted lists to produce the output, incurring the cost of random index lookups along the way. In contrast, DIL sequentially scans the inverted lists, and is likely to be faster. In general, the overhead of performing random index lookups in RDIL can sometimes outweigh the benefit of processing the inverted lists in rank order.

The above discussion presents a dilemma – both DIL and RDIL are likely to significantly outperform each other, but they require the inverted lists to be sorted in different orders. Can we combine the benefits of DIL and RDIL without replicating the entire inverted list index? We now present a hybrid technique that combines the benefits of DIL and RDIL with only a modest increase in space.

4.4.1 HDIL: Data Structure

The key idea behind HDIL is as follows. RDIL is likely to outperform DIL only if it scans a small fraction of the full inverted list; consequently, we can store the full inverted list sorted by Dewey id (for DIL), and store only a small fraction of the inverted list sorted by rank (for RDIL). Figure 9 illustrates this structure.

4.4.2 HDIL: Query Processing

Ideally, given a keyword query $k_1, …, k_n$, it will be good to make an *a priori* decision as to whether RDIL is likely to outperform DIL or vice-versa, and choose the faster alternative. However, as mentioned above, the performance of RDIL strongly depends on the keyword correlation, and such information is difficult to obtain *a priori*. Note that it is impractical to pre-compute correlations of all keyword combinations because there are too many such combinations. Since most keyword search queries are ad-hoc, pre-computing correlations for a fixed set of keyword combination will not work well either.

To address this problem, we consider an adaptive strategy. We first start evaluating the query using RDIL, and periodically monitor its performance to calculate (a) the time spent so far – *t*, and (b) the number of results above the threshold so far – *r*. Based on this, we estimate the remaining time for RDIL as $(m-r)*t/r$, where *m* is the desired number of query results. If this estimated time is more than the expected time for DIL, we switch to DIL. Note that the expected time for DIL is relatively easy to compute *a priori* for a given machine configuration because it mainly depends on the size of the query keyword inverted lists (since DIL scans inverted lists fully in all cases).

Note how the HDIL dynamically adapts to correlations. If there are very few results above the threshold (corresponding to low keyword correlation), it switches to DIL; else it sticks with RDIL.

4.4.3 HDIL: Space/Time Complexity

Besides the small overhead of monitoring performance of RDIL, the space/time complexity of HDIL is the same as DIL and RDIL.

4.5 Updating the Inverted Lists

Thus far, we have focused on querying the inverted list structures. We now briefly address the issue of updates. Document-granularity updates (i.e., adding or deleting documents) can be handled exactly like in traditional inverted lists [7][33]. The same techniques can be used because DIL, RDIL, and HDIL do not replicate ancestor information, and because the first component of the Dewey IDs contains the document ID (which can be used for deletion).

Handling the insertions of individual XML elements is more challenging because the Dewey IDs of the *siblings and descendants* of the inserted element may need to be updated (recall that Dewey IDs contain the relative position among siblings). Tatarinov et al. discuss efficient ways to update Dewey IDs under element insertions, including sparse Dewey numbering techniques. Deleting elements, however, does not require special processing.

We currently support document-granularity updates. We plan to support element-granularity updates of Dewey IDs by adapting the techniques proposed by Tatarinov et al. [31].

5. EXPERIMENTAL EVALUATION

We now experimentally evaluate the techniques presented in this paper. First, we present some anecdotal evidence that our ranking function returns intuitive results. Second, we investigate the space savings due to the Dewey encoding of element ids. Finally, we evaluate the performance of our index structures and algorithms.

5.1 Experimental Setup

We used both the DBLP and XMark data sets for our experiments. The size of the entire DBLP data set was about 143MB. We also generated a 100MB XMark data set, which corresponds to a scale factor of 1.0. We chose to experiment with the DBLP and XMark data sets for the following reasons. First, DBLP data is relatively shallow with a depth of about 4, while XMark data is relatively deep with a depth of 10. Second, DBLP data has many inter-document references (in the form of bibliographic citations), while XMark has many intra-document references (in fact, the entire XMark data set is a single XML document). Finally, DBLP and XMark represent real and synthetic data sets, respectively.

We implemented the *ElemRank* computation, and DIL, RDIL and HDIL. The inverted lists were implemented in the file system, and we built our own disk-resident B+-tree over the inverted lists for RDIL and HDIL. We initially implemented our system using a relational database, but then chose to re-implement our own inverted list and index structures for many reasons. First, the API presented by commercial B+-tree indices was not general enough to determine deepest common ancestors. Second, we found that we could not perform important space optimizations (see Section 4.3.1) on relational B+-trees. Finally, the performance using a commercial relational database system was about 5 times slower than our current implementation.

	DBLP		XMARK	
	Inv. List	**Index**	**Inv. List**	**Index**
Naïve-ID	326MB	N/A	898MB	N/A
Naïve-Rank	326MB	317MB	898MB	527MB
DIL	141MB	N/A	354MB	N/A
RDIL	141MB	150MB	354MB	320MB
HDIL	155MB	45MB	380MB	52MB

Figure 10: Space Requirements for the Different Approaches

As a baseline for comparison, we also implemented two versions of the naïve approach (Section 4.1), one where the inverted list was ordered by the ID (Naïve-ID), and another where it was ordered by rank (Naïve-Rank). Naïve-ID does a simple equality merge of the inverted lists during keyword evaluation. Naïve-Rank has a hash index built on the ID field for random equality lookups, and uses the Threshold Algorithm as a stopping condition (similar to RDIL). Note that Naïve-Rank does not need to determine longest common prefixes using B+-trees (because all ancestor IDs are explicitly stored), but only needs to determine if the same ID occurs in multiple lists. Thus, a hash-index is sufficient.

We used C++ for our implementation, and used a 1.7GHz Pentium IV processor with 1GB of main memory and 30GB of disk space. Most results shown were obtained using a cold operating system cache to simulate a non memory-resident data set. We also present some warm cache results for comparison.

5.2 Quality of Ranking Function

While a user study is beyond the scope of this paper, we present some anecdotal evidence that our keyword query semantics and ranking functions produce intuitive results. When we issued the keyword search query ‘gray’, we got both <author> elements in highly referenced papers and books written by Jim Gray, and the <title> elements of the important papers on Gray codes. This illustrates how *ElemRank* propagates rankings from highly referenced papers down to their sub-elements. When we issued the query ‘author gray’, the ranks of <title>s of Gray codes dropped due to our two-dimensional keyword proximity metric.

The keyword queries that we ran on the deeply nested XMark benchmark illustrated the benefit of returning the most specific results. For example, the keyword query ‘stained mirror’ returned an item whose name was ‘stained’ and whose description had the keyword ‘mirror’; this item was referenced by many auctions in the XMark database, and hence had a relatively high rank.

5.3 Space Requirements

Figure 10 gives the space requirements for the various approaches. As shown, the naïve approaches incur a significant space overhead for both DBLP and XMark. This is because the naïve approaches replicate ancestor IDs in inverted lists. This overhead increases with XML document depth, which explains the increased overhead for XMark. In contrast, DIL requires much less space because it only stores the IDs of leaf elements. The size of RDIL is the same as that of DIL. However, RDIL has the extra cost of storing B+-trees. Interestingly, the space overhead of B+-trees for HDIL is far less than that for RDIL; this is because the full inverted list in .HDIL is sorted by Dewey ID (see Figure 9). Hence, the inverted list can be reused as the leaf level of the B+-tree.

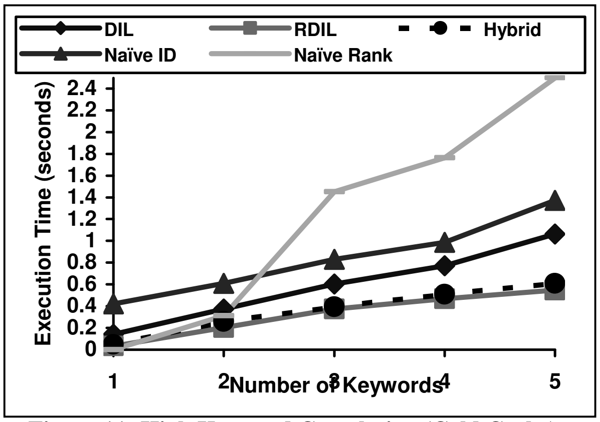

Figure 11: High Keyword Correlation (Cold Cache)

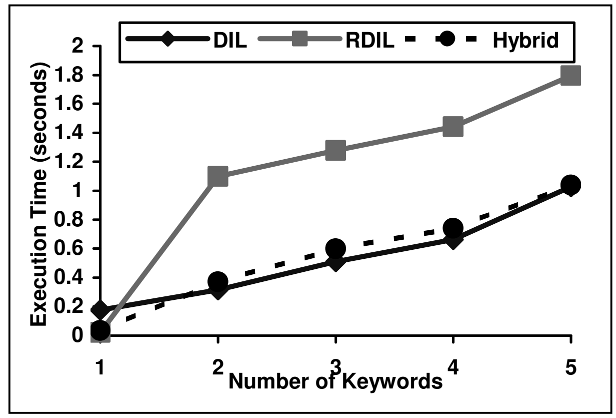

Figure 12: Low Keyword Correlation (Cold Cache)

5.4 Query Performance

We now evaluate the performance of the different approaches. There are four main factors that affect the performance of keyword search queries: (1) the *number of query keywords*; (2) the *correlation between the keywords*; (3) the desired *number of query results*; (4) the *selectivity of the keywords*. We experimented with all four parameters using both randomly generated keywords and hand-selected keywords. We found that the selectivity of the keywords is not as interesting because (a) highly selective keywords do not model large document collections, and (b) all the approaches perform about the same if the size of the inverted lists is small. We thus only consider unselective keywords here. The default value for number of query results is 10. We only report the results for the DBLP data set; the results for XMark are similar.

Figure 11 shows the performance of the different approaches when there is a high correlation between the keywords. RDIL performs well because the index probes to find common ancestors are successful. DIL, on the other hand, has to scan the entire inverted list, and hence performs relatively poorly. Note how the performance of HDIL tracks that of RDIL by estimating a low completion time for RDIL. It is also interesting to note that the performance of Naïve-ID is worse than that of DIL, and the performance of Naïve-Rank is (much) worse than that of RDIL. This is because of the extra overhead of scanning ancestor entries in the Naïve approaches. Naïve-Rank is particular bad because it also incurs the cost of random index lookups for the ancestor entries. Thus DIL, RDIL and HDIL not only save space, but also provide associated performance gains. In subsequent graphs, we do not show the performance of Naïve-ID and Naïve-Rank.

Figure 12 shows the performance of the different approaches when there is a low correlation between the keywords. Here, RDIL performs relatively badly for more than one query keyword because there are many unsuccessful random B+-tree lookups. In contrast, DIL sequentially scan the inverted lists and performs better. HDIL tracks the performance of DIL, but with a slight overhead because it starts of as RDIL, and then switches to DIL. Figure 13 shows the results for the same query on a warm cache. The results are broadly similar to the cold cache version, but are 2-3 times faster. Note also that HDIL occasionally has a running time that is slightly greater than both DIL and RDIL (for number of keywords = 2). This is because the performance of DIL and RDIL are crossing over at this point, and HDIL makes a slightly inaccurate estimation and switches to DIL instead of sticking to RDIL. We are investigating more accurate estimation techniques that will improve the prediction capabilities of HDIL in such cases.

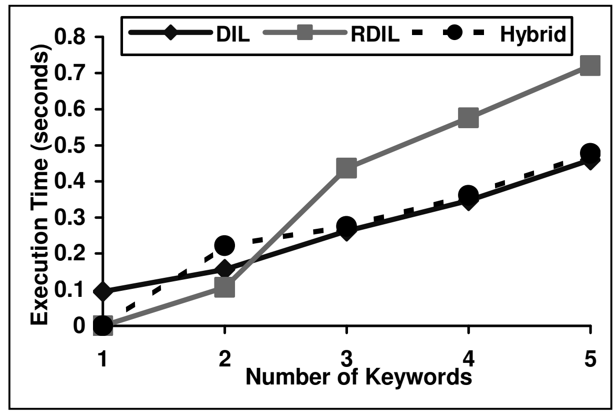

Figure 13: Low Keyword Correlation (Warm Cache)

We also varied the number of query results (not shown), and found that the performance of DIL remains about the same because it always scans the entire inverted lists. The performance of RDIL, however, decreases with an increasing query result size because RDIL has to scan more of the inverted lists.

6. RELATED WORK

There has been recent work on integrating keyword search with structured XML querying [2][5][8][15]. Schmidt et al. [29] introduce the “meet” operator for XML, which is similar to returning the most specific result. They also present efficient algorithms for computing “meet” using relational-style joins and indices. Christophides et al. [11], Dao et al. [12] and Lee et al. [25] present systems for querying structured documents. However, the above systems do not consider ranking, two-dimensional keyword proximity, rank-based query processing algorithms/inverted lists, or integration with hyperlinked HTML keyword search, all of which are central to XRANK.

The following systems support ranked XML keyword search. XIRQL [16] is an extension of XQL for information retrieval. Myaeng et al. [26] use term-occurrences to compute the ranked results over SGML documents. XXL [32] uses term occurrences and ontological similarity for ranking. Luk et al. [25] survey commercial XML search engines. We are not aware of any system that uses hyperlink structure, a two-dimensional proximity metric, specialized ranked inverted indices and query processing techniques for efficient XML and HTML keyword search.

DBXplorer [1] and DISCOVER [19] support keyword search over relational databases, but do not support information retrieval style ranking. Further, they are not directly applicable for XML and HTML documents, which cannot always be mapped to a rigid relational schema. BANKS [4], DataSpot [13] and Lore [17] support keyword search over graph-structured data. Some of these systems use hyperlinked structure (BANKS), and simple proximity (BANKS, Lore) for ranking. However, these systems do not generalize HTML search engines, and do not exploit the two-dimensional proximity inherent in XML. Further, DataSpot [13] does not present any query evaluation algorithms, and Lore [17] can only support keyword searches where the result type is known. BANKS requires that all the data edges fit in memory, which is not feasible for large data sets. Chakrabarti et al. [10] use nested HTML tag and hyperlink information to compute ranks *at the granularity of a document*. In contrast, XRANK computes rankings *at the granularity of an element* because XML keyword search queries return elements. Also, XRANK considers element-to-element links in addition to document-to-document links.

Algorithms for computing the deepest common ancestor of two nodes in a tree are well known [18], but these do not consider ranking, and are not directly applicable for lists of nodes (a naïve adaptation would require a Cartesian product of the inverted lists). Jacobson et al. [21] and Jagadish et al. [22] use Dewey IDs for hierarchical contexts and network directories, respectively. The authors also present table-driven and stack-based algorithms for checking ancestor-descendant relationships. The algorithm in Section 4.3.2 bears some similarity to these algorithms, but differs in the following ways. First, we integrate ranking during query processing. Second, we determine deepest common ancestors, which is more general than ancestor-descendant relationships. Third, we handle multi-way merges, corresponding to multiple keywords. Finally, we handle specifics of XML keyword search, such as removing spurious results and inferring position lists.

7. CONCLUSION AND FUTURE WORK

We have presented the design, implementation and evaluation of the XRANK system for ranked keyword search over XML documents. To the best of our knowledge, XRANK is the first system that takes into account (a) the hierarchical and hyperlinked structure of XML documents, and (b) a two-dimensional notion of keyword proximity, when computing the ranking for XML keyword search queries. Our experimental evaluation also shows that our specialized index structures and query evaluation techniques provide significant space savings and performance gains. XRANK is designed to naturally generalize a HTML search engine such as Google; consequently, XRANK can query over a mix of HTML and XML documents.

There are several avenues for future work. For instance, we have currently taken a document-centric view, where we assume that query results are strictly hierarchical. However, for structured (or semi-structured) data, the XML documents may be normalized, in which case the result may be a graph. Other open problems include extensions to other ranking functions (e.g., tf-idf [28]), incremental index maintenance, and integration with structured queries.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", ICDE Conf., 2002.

[2] V. Aguilera, S. Cluet, F. Wattez, "Xyleme Query Architecture", WWW Conf., 2001.

[3] V. Anc, O. de Kretser, A. Moffat, "Vector-Space Ranking with Effective Early Termination", SIGIR Conf., 2001.

[4] G. Bhalotia, et al., "Keyword Searching and Browsing in Databases using BANKS", ICDE Conf., 2002.

[5] K. Bohm, et al., "Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM", VLDB Journal 6(4), 1997.

[6] S. Brin, L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", WWW Conf., 1998.

[7] E. Brown, J. Callan, B. Croft, "Fast Incremental Indexing for Full-Text Information Retrieval", VLDB Conf., 1994.

[8] L. J. Brown, et al., "A Structured Text ADT for Object-Relational Databases", Theory and Practice of Object-Systems 4(4), 1998.

[9] C. Buckley, A. F. Lewit, "Optimization of Inverted Vector Searches", SIGIR Conference, 1985.

[10] S. Chakrabarti, M. Joshi, V. Tawde, "Enhanced Topic Distillation Using Text, Markup, Tags and Hyperlinks", SIGIR Conf., 2001.

[11] V. Christophides, et al., "From Structured Documents to Novel Query Facilities", SIGMOD Conf., 1994.

[12] T. Dao, R. Sacks-Davis, J. Thom, "An Indexing Scheme for Structured Documents and their Implementation", Conf. On Database Systems for Advanced Applications, 1997.

[13] S. Dar, et al., "DTL's DataSpot: Database Exploration Using Plain Language", VLDB Conf., 1998.

[14] R. Fagin, A. Notem, M. Naor, "Optimal Aggregation Algorithms for Middleware", PODS Conference, 2001.

[15] D. Florescu, D. Kossmann, I. Manolescu, "Integrating Keyword Search into XML Query Processing", WWW Conf., 2000.

[16] N. Fuhr, K. Grobjohann, "XIRQL: A Language for Information Retrieval in XML Documents", SIGIR Conf., 2001.

[17] R. Goldman, et al., "Proximity Search in Databases", VLDB Conf., 1998.

[18] D. Harel, H. E. Tarjan, "Fast Algorithms for finding nearest common ancestors", SIAM J. of Computing, vol. 13, 1984.

[19] V. Hristidis, Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases", VLDB Conf., 2002.

[20] HyTime, http://www.hytime.org.

[21] G. Jacobson, et al., "Focusing Search in Hierarchical Structures with Directory Sets", CIKM Conf., 1998.

[22] H. V. Jagadish, et al., "Querying Network Directories", SIGMOD Conference, 1999.

[23] J. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", JACM 46(5), 1999.

[24] Y. Lee, et al., "Index Structures for Structured Documents", Digital Libraries Conf., 1996.

[25] R. Luk, et al., "A Survey of Search Engines for XML Documents", SIGIR Workshop on XML and IR, 2000.

[26] S. Myaeng, et al., "A Flexible Model for Retrieval of SGML Documents", SIGIR Conf., 1998.

[27] M. Persin, "Document Filtering for Fast Ranking", SIGIR Conference, 1994.

[28] G. Salton, "Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer", Addison Wesley, 1989.
[29] A. Schmidt, M. Kersten, M. Windhouwer, "Querying XML Documents Made Easy: Nearest Concept Queries", ICDE Conf., 2001.
[30] A. Schmidt, et al., "The XML Benchmark Project", Tech. Report INS-R0103, CWI, The Netherlands, 2001.
[31] I. Tatarinov, et al., "Storing and Querying Ordered XML Using a Relational Database", SIGMOD Conf., 2002.
[32] A. Theobald, G. Weikum, "The Index-Based XXL Search Engine for Querying XML Data with Relevance Rankings", EDBT Conf., 2002.
[33] A. Tomasic, H. Garcia-Molina, J. Schoens, "Incremental Updates of Inverted Lists for Text Document Retrieval", SIGMOD Conf., 1994.
[34] World Wide Web Consortium, http://www.w3.org.