Context-Sensitive Keyword Search and Ranking for XML

Chavdar Botev
Cornell University
cbotev@cs.cornell.edu

Jayavel Shanmugasundaram
Cornell University
jai@cs.cornell.edu

1. INTRODUCTION

Traditionally, keyword-search-based information retrieval (IR) has focused on “flat” documents, which either do not have any inherent structure or have structure that is not exploited by the IR system. Thus, even if users wanted to search over only specific sub-sets and/or sub-parts of the documents, they still had to search over the entire document collection. In contrast, many emerging XML document collections have a hierarchical structure with semantic tags, which allows users to specify the context of their search more precisely.

As an illustration, consider a large and heterogeneous XML digital library that contains content ranging from Shakespeare's plays to scientific papers. A user who is interested in learning more about Shakespeare’s plays can limit the scope of her search to just the relevant plays by specifying the following XPath query: //Play[author = ‘William Shakespeare’]. Thus, if she were to issue a keyword search query for the words “speech” and “process”, she would only get XML element results in Shakespeare’s plays that contain the keywords (such as a relevant <speech> element), and would not get XML elements about (say) voice recognition systems.

In this paper, we refer to this notion of restricting the search context as context-sensitive search. Supporting context-sensitive search introduces the following two challenges. The first challenge is to *efficiently find search results in the search context without having to touch irrelevant content.* In the example above, if Shakespeare’s plays constitute only 0.1% of the entire content of the digital library, an efficient context-sensitive search implementation should not process the remaining 99.9%. The second challenge is to *effectively rank keyword search queries evaluated in a search context.* For example, in the popular TF-IDF scoring method, the IDF component represents the inverse document frequency of the query keywords in the entire document collection. However, using this IDF value directly for context-sensitive search can produce very unintuitive results.

As an illustration, consider the keyword search query for the words “speech” and “process” over Shakespeare’s plays in XML [20]. We obtained the top 10 results from the query using one of the TF-IDF based XML ranking algorithms [1] published in the literature. We then took a heterogeneous XML collection consisting of IEEE INEX 2003 documents [11] (containing scientific papers) and Shakespeare’s plays in XML, limited the search context to Shakespeare’s plays, evaluated the same keyword search query and obtained the top 10 results using the same ranking algorithm. From the user’s point of view, these are semantically identical queries. However, 9 of the top 10 results in the first set of results were not present in the second set!

Copyright is held by the author/owner.
Eighth International Workshop on the Web and Databases (WebDB 2005),
June 16-17, 2005, Baltimore, Maryland.

This wide variation in results can be explained as follows. It turns out that “speech” is very frequent in Shakespeare’s plays (low IDF) but not in the entire collection (high IDF), while “process” is relatively frequent in the entire collection (low IDF) but not in Shakespeare’s plays (high IDF). Consequently, the first experiment emphasized results that contained “process”, while the second emphasized results that contained “speech”. From the user’s point of view, the first experiment is likely to return more meaningful results because “process” – and not “speech” – is the more uncommon word in the search context (Shakespeare’s plays). Thus, the results of the second experiment are heavily skewed by elements that are not even in the search context.

In general, it is desirable for context-sensitive search results to be ranked as though the user query was evaluated over the search context in isolation (in our example, we desire that the second experiment should return the same results as the first experiment). We call this *context-sensitive ranking* (a similar concept is also referred as query-sensitive scoring in [7]).

In this paper, we present the design, implementation and evaluation of a system that addresses the above issues central to context-sensitive search. We make the following contributions:

1) We present enhanced inverted list structures and query evaluation algorithms that enable the efficient evaluation of context-sensitive keyword-search queries, without having to touch content irrelevant to the search context.
2) We develop a framework that allows for efficient context-sensitive ranking. We note that, unlike [7], our goal is not to develop new ranking algorithms for context-sensitive search. Rather, our goal is to provide a framework that will enable existing ranking methods to be used for context-sensitive ranking.

We also quantitatively evaluate the performance of our inverted list data structure and context-sensitive ranking based on existing XML ranking algorithms.

2. SYSTEM MODEL AND ARCHITECTURE

2.1 Model

We represent a collection of XML documents as a forest of trees $G = (N, E)$, where N is the set of XML element nodes and E is the set of containment edges relating vertices. Node u is a *parent* of a node v if there is an edge $(u, v) \in E$. Node u is an *ancestor* of a node v if there is a sequence of one or more containment edges

that lead from u to v. The predicate *contains*(v, k) is true iff the vertex v directly or indirectly (through sub-elements) contains the keyword k, where k can be an element name, attribute name, attribute value, or other textual content. The granularity of query results is at the level of XML *elements* since returning specific elements (such as <speech>) usually gives more context information than returning the entire document ([1][5][8][21]).

2.2 System Architecture

Figure 1 shows our system architecture. The user query consists of two parts: (1) the keyword search query, and (2) the search context. Given a specification of the search context, the Context Evaluator returns a set of XML element IDs I such that the descendants of I define the search context. The search context consists of all the elements in the sub-trees rooted at the elements from the specification value. The Query Engine takes in the set of IDs I and the user keyword-search query, and ranks the elements in the search context with respect to the query. In producing the ranked query results, the Query Engine uses Index Structures and a Ranking Module. The latter is extensible with respect to various ranking functions.

For ease of exposition, we assume simple disjunctive queries (i.e., queries using only the 'or' operator) in this paper. The search context for these queries is specified using XPath. We use a standard XPath evaluator leveraging the work in [22].

The focus of this paper is on the Index Structures and Query Engine components. In the following section, we address three main challenges in designing these components. First, we show how to efficiently limit the search results to only those elements that occur in the search context. Second, we design a framework that supports context-sensitive ranking, i.e., ranking as though the query was evaluated over the search context in isolation. Third, we present an efficient evaluation algorithm that integrates these two solutions.

3. INDEXING AND QUERY EVALUATION

We first define the problem of context-sensitive ranking. We then describe our index structures, our ranking framework, and its integration with the query engine.

3.1 Context-Sensitive Ranking

Consider a set of XML elements E, a ranking algorithm R, and a keyword-search query Q. We define $RankRes_{E,R,Q}$ to be the set of pairs (e, s), where $e \in E$ and s is the score (rank) of e with respect to the query Q obtained using the ranking algorithm R. Intuitively, $RankRes_{E,R,Q}$ is the ranked results that we would have obtained if E was a stand-alone collection and the query results were ranked using R. Now, consider a context-sensitive search system S that uses the ranking algorithm R and is operational over an element collection E. Consider a user query (Q, SC), where Q is the keyword-search query, and $SC \subseteq E$ is the set of elements that constitute the user-specified search context. We define $CRankRes_{S,E,SC,R,Q}$ to be the set of pairs (e, s) returned by S for the user query (Q, SC), where $e \in SC$ and s is the score of e obtained using R in the SC.

We say that a system S supports *context-sensitive ranking* with respect to a ranking algorithm R iff for every set of XML elements E and for every user query (Q, SC) (where $SC \subseteq E$),

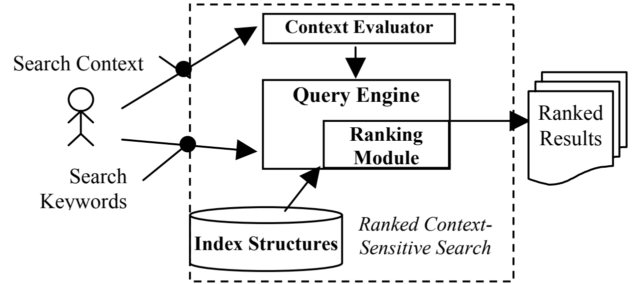

Figure 1: System Architecture

$RankRes_{SC,R,Q}$ = $CRankRes_{S,E,SC,R,Q}$. In other words, the ranked results produced by S for a user query (Q, SC) are exactly the same as the ranked results produced in a stand-alone collection SC (using the ranking algorithm R). Thus, the system S provides users with the abstraction of working with a personalized document collection defined by the search context (SC), even though the search context may be part of a large heterogeneous collection (E) in the system. This will avoid ranking anomalies such as the one reported in the introduction.

We note that our focus is *not* on the design of new ranking algorithms R. Rather, our goal is to develop a general framework so that many *existing* ranking algorithms can be embedded in our system while still supporting context-sensitive ranking.

3.2 Index Structure

Our main goal is to enable the efficient evaluation of context-specific queries. A naïve strategy is to evaluate the user query over all the elements in the index, and check whether each element is present in the search context. This approach, however, is likely to be very inefficient when the search context is only a very small fraction of the entire collection.

We now propose an index structure that addresses the above issues by extending the inverted list index structure [18]. Two key modifications need to be made to make the inverted list applicable for context-sensitive XML keyword search. First, we need to capture the XML hierarchy in the inverted list entries so that nested elements that contain the query keywords can be returned as results; for this part, we build upon prior published work in this area [8][13]. Second, we need to structure the inverted list so that entries that do not belong to the search context can be easily skipped; this will enable the efficient evaluation of context-specific queries. We consider each in turn.

3.2.1 Capturing XML Hierarchy in Inverted Lists

A simple way to structure the inverted list is to store for each keyword the IDs of all elements that *directly or indirectly* contain the keyword. The downside of this approach is the associated space overhead. We need to store the IDs of the elements that directly contain the keyword *and the IDs of their ancestors*, because the ancestors too *indirectly* contain the keyword and should be returned as query results [8].

To address this space (and performance) overhead, Guo et al. [8] and Lee et al. [13] propose an encoding for element IDs called Dewey IDs. Each element is assigned a number that represents its relative position among its siblings in the XML document tree. The

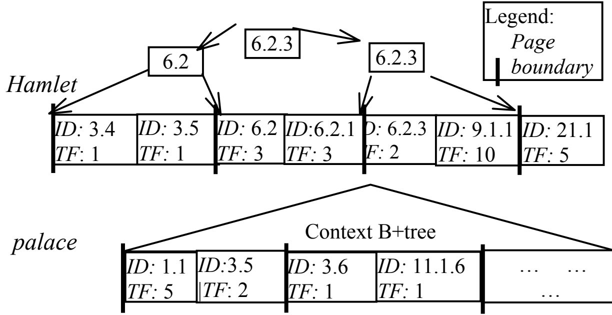

Figure 2: Inverted List Structure

path vector of the numbers from the root to an element uniquely identifies the latter. For example, the top element will be assigned Dewey ID 1; its children will be assigned 1.1, 1.2, …, etc.

Thus, ancestor-descendant relationships are implicitly captured and the inverted lists only need to store the IDs of the elements that *directly* contain the keyword. Figure 2 shows a fragment of the inverted list index using Dewey IDs for the words "Hamlet" and "palace" (ignore the page boundaries and B+-trees for now – they are used for context-sensitive search, discussed next). For each keyword, the entries contain the IDs of the XML elements that directly contain the keyword and other information such as the TF for the elements.

3.2.2 Efficiently Limiting the Search Context

To limit efficiently the search context *SC* for a query *Q*, we need to skip over the entries in the inverted list that do not belong to the search context. Also, recall from the system architecture that the Context Evaluator returns element IDs and all *descendants* of these elements constitute the search context; thus, we also need to identify the descendants efficiently.

It turns out that both of these problems can be addressed elegantly by building a B+-tree index on each inverted list based on the Dewey ID. Since the inverted lists are sorted based on the Dewey ID, the inverted lists can serve as the leaf level of the B+-tree; this avoids having to replicate the inverted list in the B+-tree, and leads to large space savings. Figure 2 shows this a sample index organization with fan-out 2.

The B+-tree index can be used to efficiently skip over irrelevant entries as follows. Assume that the Context Evaluator returns a list of Dewey IDs d_1, …, d_n that define the search context. We first start with d_1 (say 6.2) and probe the B+-tree of the relevant keyword inverted list to determine the smallest ID in the inverted list that is greater than or equal to d_1 in lexicographic order (6.2 in our example). We then start sequentially scanning the inverted list entries from that point onwards. Since all descendants of d_1 are clustered immediately after d_1 in the inverted list (as they share a common prefix), this scan will return all the descendant entries of d_1 (6.2, 6.2.1, 6.2.3). The scan will be stopped as soon as an entry is encountered whose prefix is not d_1 (and hence not a descendant of d_1). The same process is repeated for d_2, …, d_n. Note how all possible descendants of d_1 (e.g., 6.2.2,6.2.2.1, etc.) are *not* explicitly enumerated but only the descendants that appear in a relevant inverted list are retrieved.

3.3 Ranking Module

There are three important aspects that we consider in the design of the Ranking Module. First, it should support context-sensitive ranking, whereby only the elements that belong to the search context contribute to the ranks of the query results. Second, the Ranking Module should provide an extensible framework that supports a general class of ranking functions so that many existing (and possibly new) ranking schemes developed for non-context-sensitive ranking of XML results can be directly applied to the context-sensitive ranking problem. Finally, the Ranking Module should enable tight integration with the Query Engine so that context-specific ranking can be done efficiently.

In the following discussion, we will focus on TF-IDF based ranking methods since they are one of the most popular scoring methods used in IR. Furthermore, these scoring methods are well suited for presentation of the concepts of context-sensitive ranking. Indeed, the IDF value depends on the search context: it is the number of *search context elements* that contain a query keyword. Thus, such ranking methods can dynamically adapt based on whether the query keywords are frequent or rare in the search context (irrespective of whether they are frequent or rare in the entire collection). We note that the TF component depends on the content of an element and is usually independent of the context.

3.3.1 Modeling XML Ranking Functions

Given a user keyword search query k_1, …, k_n, issued over a search context *SC*, most TF-IDF based XML ranking methods [1][5][21] can be characterized as a function $\boldsymbol{R}(C_{k1}, \ldots, C_{kn}, E_{k1,e}, \ldots, E_{kn,e})$ that takes in the following parameters for each element $e \in SC$ and returns the score for *e*.

- $C_{k1}, \ldots, C_{kn}$: Each C_{ki} contains scoring information based on a query keyword k_i and the search context *SC* (e.g. the IDF for k_i)
- $E_{k1,e}, \ldots, E_{kn,e}$: Each $E_{ki,e}$ contains scoring information based on a keyword k_i the element *e*, and its descendants (e.g. TF of the keyword k_i with respect to *e*).

As an illustration, consider the XXL search engine [21] that uses the TF-IDF ranking method. The C_{kj} parameters are the element frequency values for the keyword k_j: C_{kj} = <number of elements containing k_j>/<number of elements in the search context> = $1/idf(k_j)$. The $E_{kj,e}$ parameters contain the normalized TF for the keyword k_j with respect to the element *e*: $E_{kj,e}$ = <number of occurrences of k_j in *e*>/<maximum term occurrences in *e*>. The overall-score function $\boldsymbol{R}$ combines the C_{kj} and the $E_{kj,e}$ parameters using the cosine similarity: $\boldsymbol{R}(C_{k1}, \ldots, C_{kn}, E_{k1,e}, \ldots, E_{kn,e})$ = $\sum_{j=1,..,n} \frac{(E_{kj,e}/C_{kj}) \times idf(k_j) tf(k_j, Q)}{\|e\|_2 \times \|Q\|_2}$, where $\|\cdot\|_2$ denotes the L_2 measure. The XSearch [1] and XIRQL [5] ranking methods can be similarly captured using the above framework.

3.3.2 Integration with the Index and Query Engine

For efficient query evaluation, our query processing algorithm (described in Section 3.4) relies on two fundamental principles: (1) During the evaluation of a keyword search query k_1, …, k_n, only the inverted list entries for the keywords k_1, …, k_n, that occur in the search context are accessed. No other inverted list entries

are accessed, nor is the actual text content of an element accessed during query processing; (2) Query evaluation occurs in a single pass over the query keyword inverted lists (although a pre-processing pass is also used – details are in Section 3.4). Thus, once an index entry is accessed, it is not accessed again.

The above observations suggest that the ranking function parameter values should (1) be computed solely from the query keyword index entries in the search context, and (2) should be accumulated in a single pass over these index entries. We now formalize these notions. Consider the computation of a C_{kj} parameter. It can be computed using a function $\boldsymbol{FC_{kj}}$ that works like an accumulator: $\boldsymbol{FC_{kj}}$: $\mathrm{Dom}(C_{kj}) \times \mathit{InvListEntry} \rightarrow \mathrm{Dom}(C_{kj})$, where $\mathrm{Dom}(C_{kj})$ is the domain of the values of C_{kj}. On each invocation, it takes in the current value of C_{kj} and the current index entry, and creates the new value of C_{kj}. When the last entry is processed, the result is the final value of the C_{kj} parameter. For example, consider the case of computing the IDF of the keyword k_j. The initial value of C_{kj} is 0, and each invocation of the $\boldsymbol{FC_{kj}}$ function simply increments the current value of C_{kj} by one over the number of elements in the search context. The IDF is the reciprocal of the final value of C_{kj} after processing all entries in the inverted list for k_j that occur in the search context.

Now, consider the computation of a $E_{kj,e}$ parameter. $E_{kj,e}$ is computed by the repeated application of the function $\boldsymbol{FE_{kj}}$: $\mathrm{Dom}(E_{kj,e}) \times \mathit{Node} \times \mathit{Node} \times \mathit{InvListEntry} \rightarrow \mathrm{Dom}(E_{kj,e})$. The function works like an accumulator and takes in the current value of $E_{kj,e}$ (first parameter), the current search context element *e* (second parameter), a descendant of *e* containing directly k_j (third parameter), and the index entry corresponding to the descendant (fourth parameter), and uses these arguments to compute the new value of the $E_{kj,e}$ parameter. This function captures the intuition that only information in the inverted list corresponding to an element's descendants is used to compute element-specific ranking information. For example, in XXL, the $\boldsymbol{FE_{kj}}$ function updates the TF of the keyword k_j with respect to the current search context element *e*.

3.4 Query Engine

We now describe how the Query Engine can efficiently support context-sensitive ranking for any ranking algorithm that can be characterized in terms of the $\boldsymbol{R}$, $\boldsymbol{FC_{kj}}$, and $\boldsymbol{FE_{kj}}$ functions (by efficient, we mean linear in the number of index entries in the search context). Our query-processing algorithm builds upon the work in [8] and extends it using a two-phase algorithm for context-sensitive search and ranking. In the first (pre-processing) phase, it computes context-sensitive information that is used for ranking (i.e., the C_{ki} or IDF values) by making a pass over the relevant parts of the query keyword inverted lists. In the second (regular) phase, it makes another pass over the same inverted list entries, and finds the top-k ranked query results.

The pre-processing phase is necessary because the regular phase cannot compute the overall rank of an element without the appropriate context-sensitive values. Thus, the regular phase cannot just keep track of the top-k results using a result heap since the score cannot be computed until the very end. Consequently, all elements should be retained and scored, which will be expensive. We note that the overhead of an extra phase is minimal since the first phase will bring all the relevant disk resident entries into memory; thus, the second phase, which accesses the very same entries, has practically no overhead (see Section 4).

```
01. procedure EvaluateQuery (k1, k2, …, kn, cid1, …, cidm, N)
    // k1 … kn are query keywords, cid1 … cidm define search context,
    // N is the # query results  invList[ki] is inverted list for ki,
    // btree[ki] is context B+-tree for ki

    // Pre-processing phase: compute Ckis
02. for (each cidj) {
03.   for (each ki) {
04.     ilPos = btree[ki].probe(cidj); invList[ki].startScan(ilPos);
05.     curEntry = invList[ki].nextEntry;
06.     while (cidj is a prefix of curEntry.deweyID) {
07.       Cki = FCki(Cki, curEntry);
08.       invList[ki].nextEntry;
09.} } }

    // Regular phase; compute top-N query results
10. resultHeap = empty; deweyStack = empty;
11. for (each cidj) {
12.   for (each ki) {
13.     ilPos = btree[ki].probe(cidj); invList[ki].startScan(ilPos);
14.     curEntry[ki] = invList[ki].nextEntry;
15.   }
16.   while (∃ki such that cidj is a prefix of curEntry[ki]) {
        // Get the next inverted list entry with the smallest DeweyID
17.     find ki such that curEntry[ki].deweyID is the smallest deweyID;

        // Find the longest common prefix between deweyStack
        // and currentEntry.deweyId
18.     find largest lcp such that
          deweyStack[p] = curEntry[ki].deweyId[p], 1 <= p <= lcp

        // Pop non-matching deweyStack entries
        // (their descendants have been fully processed)
19.     while (deweyStack.size > lcp) {
20.       sEntry = deweyStack.pop();
21.       score = R(Ck1, …, Ckm, sEntry.Ek1, …, sEntry.Ekm);
22.       if score among top N seen so far,
            add (deweyStack.deweyID, score) to resultHeap;
23.     }

        // Push new ancestors (non-matching part of
        //currentEntry.deweyId) to deweyStack
24.     for (all i such that lcp < i <= currDeweyIdLen)
25.       { deweyStack.push(deweyStackEntry); }

        // Accumulate inverted list score information
26.     for (each deweyStack entry sEntry) {
27.       sEntry.Cki = FEki(sEntry.Cki, sEntry.deweyID,
                    currentEntry.deweyID, currentEntry);
28. }  } // End of looping over all inverted lists

29.   Pop entries of deweyStack in context cidj, and add to result heap
      (similar to lines 19-23)
30. } // End of processing cidj
31. return resultHeap
```

Figure 3: Query Algorithm

Figure 3 shows the query evaluation algorithm. The pre-processing phase (lines 02–09) works as follows. For each the search context ID cid_j, and for each query keyword k_i, it identifies the entries in the inverted list for k_i that are descendants of cid_j. It does this by probing the context B+-tree for keyword k_i using cid_j, and scanning the inverted list for k_i from that point onwards. The

Push		*Pop*		*Push*		*Update*	
4	(1, 0)			5	(1, 0)	5	(1, 2)
3	(1, 0)	3	(1, 0)	3	(2, 0)	3	(2, 2)
ID	**(E_1, E_2)**	**ID**	**(E_1, E_2)**	**ID**	**(E_1, E_2)**	**ID**	**(E_1, E_2)**
(*a*)		(*b*)		(*c*)		(*d*)	

Figure 4 The DeweyStack Transition

context-sensitive information C_{ki} is accumulated for each entry.

The second phase (lines 10-31) computes the top-k query results using the context-sensitive information C_{ki}. For efficiency, it maintains a stack of Dewey IDs, the DeweyStack. Using the DeweyStack, we can keep track of the score information of both the elements in the inverted lists and their *ancestors* (since the ancestors also indirectly contain the query keywords; note that this dependence is explicitly captured by the $\boldsymbol{FE_{ki}}$ function). The scoring information for the ancestors is updated while a descendant is being processed. This can be achieved using the DeweyStack because all ancestors and descendants are clustered in the B+-tree.

The algorithm for the second phase works as follows. For each cid_j, the relevant index entries for all the query keywords are scanned in parallel until the end of the current context (lines 16 - 28). The smallest ID from these lists is chosen (line 17). Based on it, the algorithm identifies the entries in the DeweyStack whose descendants have been fully processed; these entries are popped out of the stack, their ranks are computed using the function $\boldsymbol{R}$, and they are added to the result heap if they are among the top-k results so far (lines 19-23). The ancestors of the current smallest ID are then pushed onto the stack (lines 24-25), and the score information of all these ancestors is updated based on the current index entry (using function $\boldsymbol{FE_{ki}}$). This process is repeated until all of the relevant entries from the inverted list are processed.

As an illustration, consider the keyword query 'Hamlet palace' over the search context defined by the ID 3 using the index in Figure 2. The first phase computes C_{ki}'s (IDFs) by accessing the relevant entries: 3.4 and 3.5 for "Hamlet", and 3.5 and 3.6 for "palace". Then, in the second phase, the relevant entries are merged. First, the entry 3.4 is processed because it has the smallest DeweyID (Figure 4a). The stack state keeps track of the current score (TF value) for both 3.4 and all of its ancestors (3, in our case). The next entry processed is 3.5 from the first inverted list. Since the largest common prefix with the DeweyStack entry is 3, we can conclude that all descendants of 3 in the stack (3.4) do not have any further descendants in the search context. Thus, 3.4 is popped from the stack and is added to the result heap if it is one of the current top-k results (Figure 4b). Next, 3.5 from the first inverted list is pushed onto the stack (Figure 4c), and then 3.5 from the second list is used to update the TF values in the stack (Figure 4d). The algorithm then reads in 3.6, pops out 3.5, and continues in a similar manner.

4. EXPERIMENTAL RESULTS

We have implemented the system framework and algorithms described in the previous sections using C++. Using this framework, we have implemented context-sensitive versions of the following: XXL [21], XSEarch [1], and XIRQL [5]. We indexed a heterogeneous XML collection consisting of Shakespeare's plays [20], INEX IEEE articles [11], and SIGMOD Record in XML [19]. The size of the entire collection was 521MB, and the size of the inverted lists was 719MB. The space overhead for the context B+-trees to enable context-sensitive search was just 12MB. Our experiments were performed on a Pentium IV 2.2GHz processor with 1GB of RAM running Windows XP. When measuring performance, we used a cold operating system cache.

We performed two types of experiments. The first type measured the performance benefits of using context B+-trees to skip over irrelevant entries in the inverted lists. The second type of experiment measured how much context-sensitive ranking can influence the ranks of query results.

For the first set of experiments, we compared the performance of the following three implementations: (1) a baseline naïve approach that scans all the entries in the inverted lists, including those that do not belong to the search context (*Naïve*), (2) the algorithm in Section 3.4, but without using the pre-processing phase to compute the context-sensitive C_{ki} (IDF) values (*CSS*), (3) the full context-sensitive search and ranking algorithm described in Section 3.4 (*CSSR*). *CSS* only supports context-sensitive search, but does not support context-sensitive ranking. Thus, the performance difference between *CSSR* and *CSS* quantifies the performance overhead of context-sensitive ranking.

Figure 5 shows the performance results when the size of the search context (the percentage of the total number of elements that are in the search context) is varied. This suggests that context-sensitive search offers significant performance benefits (by up to a factor of 5) over *Naïve*. The latter does not skip over irrelevant entries in the inverted list. In contrast, CSS and CSSR show consistently better performance with smaller context sizes because they only have to scan the relevant portions of the inverted lists. We expect this difference to be even bigger for larger databases or more selective keywords.

Interestingly, there is practically no overhead for *CSSR* as compared to *CSS*, even though *CSSR* makes two passes over the relevant entries. The reason is that the first pass of *CSSR* brings all the relevant entries into memory; hence, the second scan has no measurable overhead. This suggests that context-sensitive ranking adds no measurable overhead.

The second experiment compared two lists of ranked results produced by the same ranking algorithm. The first list was the top-10 results when the IDF value was computed using the entire collection. The second set was the top-10 results produced when the IDF value was computed using only the search context elements (which is ideally what the user would like to see). The difference between these two lists is thus a measure of how much context-sensitive ranking is likely to change what the user sees in the top ranked results.

Table 2 shows the scaled Spearman Footrule Distance [2] as a measure of the difference for the XXL, XSEarch and XIRQL ranking methods for some search contexts and keywords where there are high variations in the IDF values. The scaled Footrule distance measure produces values in the range [0, 1], where 0 means identical results and 1 means that the ranked lists do not have common elements. As shown, some query keywords such as "process speech" have almost no common results in the two lists, while others have more common results.

In summary, the experiments show that context-sensitive ranking

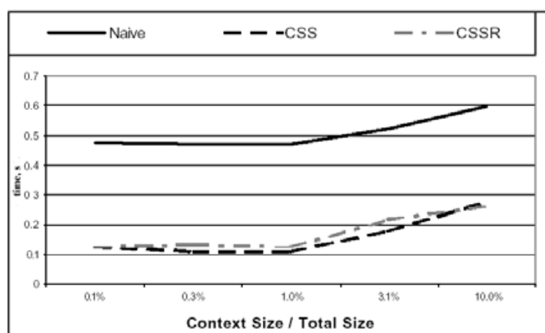

Figure 5 Context Size vs. Query Time

can significantly influence ranked results with negligible performance overhead.

5. RELATED WORK

There has been a lot of recent work on keyword search over XML. Some of these, like [3], [10], [24] and the SGML indexing techniques in [13], do not consider the issue of ranking. Various scoring methods for semi-structured document collections have been proposed [1][4][5][8][16][17][21][23]. However, unlike the present paper, none of the above addresses the issue of context-sensitive ranking and its tight integration with context-sensitive search.

Grabs and Schek [7] propose a context-sensitive scoring method for the INEX collection. Their definition of context uses predefined categories (element nodes of the same type). Our work is complementary to the above work in that we do not propose a specific scoring method but develop a general framework whereby multiple scoring methods, including that in [7], can be incorporated. Our focus is thus on developing the underlying system architecture, efficient inverted lists and query evaluation using these inverted lists, which are not considered in [7]. We also support a more flexible search context specification based on XPath without restrictions on the search context. Halverson et al. [9] and Kaushik et al. [15] discuss inverted lists with B+-trees in the context of structural joins, but do not consider context-sensitive ranking. Jacobson et al. [12] propose techniques for context-sensitive search over LDAP repositories but they focus on efficiently evaluating the context expression and not on evaluating keyword-search queries or ranking results.

6. CONCLUSIONS

We have defined the problem of context-sensitive ranking and studied its integration with context-sensitive search. We have proposed a general ranking framework whereby a large class of existing TF-IDF based ranking algorithms can be directly adapted for context-sensitive ranking. We have also proposed efficient index structures and query evaluation strategies for evaluating and ranking context-sensitive queries. In the future, we plan a user evaluation study to quantify the retrieval benefits of context-sensitive ranking.

Table 1 Effect of the Context-Sensitive Ranking

Query	Context	XXL	XSEarch	XIRQL
process speech	shakesp	0.81	0.81	1
join complexity	inex/tk	1	0.81	0.02
Sigmod opportunities	inex/tk	0.59	0.19	0.8
itemsets statistics	inex/tk	1	0.51	0.13
relational decomposition	inex/tk	1	0.32	0.16

7. REFERENCES

[1] Cohen, S., J. Mamou, Y. Kanza, Y. Sagiv. XSEARCH: A Semantic Search Engine for XML. VLDB 2003.

[2] Diaconis, P., R. L. Graham. "Spearman's Footrule as a Measure of Disarray". *J. of the Royal Society of Statistics*, series B39 (1977).

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

[4] Fuhr, N., T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. TOIS, 15 (1), 1997.

[5] Fuhr, N., Großjohann, K. XIRQL: A Language for Information Retrieval in XML Documents. SIGIR 2001.

[6] Goldman R., N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina. Proximity Search in Databases. VLDB 1998.

[7] Grabs, T., H.-J. Schek. "PowerDB-XML: A Platform for Data-Centric and Document-Centric XML Processing". XSym 2003, Berlin, Germany.

[8] Guo, L, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

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

[10] Hristidis, V., Y. Papakonstantinou, A. Balmin. Keyword Proximity Search on XML Graphs. ICDE 2003

[11] INEX 2003. http://inex.is.informatik.uni-duisburg.de:2003/

[12] Jacobson, G., B. Krishnamurthy, D. Srivastava, D. Suciu. Focusing Search in Hierarchical Structures with Directory Sets. CIKM 1998.

[13] Lee, Y., S.-J. Yoo, K. Yoon, P. B. Berra. Index Structures for Structured Documents. Digital Libraries Conf., 1996.

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

[15] Kaushik, R., R. Krishnamurthy, J. Naughton, R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. SIGMOD 2004.

[16] Myaeng, S., D.H. Jang, M.S. Kim, and Z.C. Zhoo. A Flexible Model for Retrieval of SGML Documents. SIGIR 1998.

[17] Navarro, G., Baeza-Yates, R. Proximal Nodes: A Model to Query Document Database by Content and Structure. Information Systems, 1997.

[18] Salton, G. Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer. Addison Wesley, 1989.

[19] SIGMOD Record in XML. http://www.acm.org/sigmod/record/xml/XMLSigmodRecordNov2002.zip

[20] Shakespeare's Plays in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html

[21] Theobald, A., Weikum, G. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002.

[22] Yoshikawa, M., T. Amagasa, T. Shimura, S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM TOIT 1(1), 2001.

[23] Yu, C., Qi, H., Jagadish, H. Integration of IR into an XML Database. INEX Workshop, 2002.

[24] C. Zhang, J. Naugthon, D. DeWitt, Q. Luo, G. Lohman. "On Supporting Containment Queries in Relational Database Management Systems". SIGMOD 2001