Supporting Top-K Keyword Search in XML Databases

Liang Jeff Chen, Yannis Papakonstantinou

Department of Computer Science and Engineering , UCSD
La Jolla, CA, US
{jeffchen,yannis}@cs.ucsd.edu

***Abstract*— Keyword search is considered to be an effective information discovery method for both structured and semi-structured data. In XML keyword search, query semantics is based on the concept of *Lowest Common Ancestor* (LCA). However, naive LCA-based semantics leads to exponential computation and result size. In the literature, LCA-based semantic variants (e.g., ELCA and SLCA) were proposed, which define a subset of all the LCAs as the results. While most existing work focuses on algorithmic efficiency, top-K processing for XML keyword search is an important issue that has received very little attention. Existing algorithms focusing on efficiency are designed to optimize the semantic pruning and are incapable of supporting top-K processing. On the other hand, straightforward applications of top-K techniques from other areas (e.g., relational databases) generate LCAs that may not be the results and unnecessarily expand efforts in the semantic pruning. In this paper, we propose a series of join-based algorithms that combine the semantic pruning and the top-K processing to support top-K keyword search in XML databases. The algorithms essentially reduce the keyword query evaluation to relational joins, and incorporate the idea of the top-K join from relational databases. Extensive experimental evaluations show the performance advantages of our algorithms.**

I. Introduction

Keyword search is considered to be an effective information discovery method for structured and semi-structured data [1], [2], [3], [4], [5], [6], [7]. It allows users without prior knowledge of schema and query languages to search. In XML keyword search, the results of a keyword query are no longer entire XML documents, but instead are XML elements that contain all the keywords. The intuition is that keywords may be found over multiple elements. The LCA of these elements contains all the keywords and thus can be a result. Consider the keyword query {XML, data} over the XML document of Figure 1. Nodes 1.1.2.2.1 and 1.1.2.3.2 contain the two keywords, and node 1.1.2 is their lowest common ancestor. So the subtree rooted at 1.1.2 contains all the keywords and is expected to be the result.

The naive LCA-based semantics is straightforward, but leads to exponential computation and result size. Consider two lists of nodes $L_{xml} = \{u_1, u_2, \ldots, u_m\}$ and $L_{data} = \{v_1, v_2, \ldots, v_n\}$ containing two keywords {XML} and {data} respectively. For any pair of $u_i, i \in [1, m]$ and $v_j, j \in [1, n]$, there exists an LCA for them in the XML tree. In other words, for this two-keyword query, the total number of the LCAs is $m \times n$. More generally, for the naive LCA-based semantics, the result size is exponential to the query size, though many pairs may share the same LCA.

This research was supported by NSF IIS award 0713672

The several LCA-based semantic variants that have been proposed specify a subset of the $m \times n$ LCAs as the results. The most widely followed variants are ELCA [5], [8] and SLCA [6], [9], [10], [11], [12]. The algorithmic challenge of the semantic variants is to achieve the *pruning* without computing all the LCAs. Most existing work [5], [8], [6], [9], [11] focuses on this topic, addressing how to answer keyword queries efficiently. The main idea is utilizing the document order of XML elements to pre-prune LCAs so that result candidate space is largely reduced.

While query semantics and algorithm efficiency have been widely discussed, top-K keyword search in XML databases is an important issue that very little work has concentrated on. As is typical in the keyword search systems, a ranking function can be defined [5], [13] to assign to results ranking scores, and ranked results are returned to users. Top-K processing aims to compute the results with highest scores first so that execution can terminate earlier after the top K results have been generated.

Existing algorithms focusing on efficiency cannot provide effective support for top-K processing. These algorithms share some common characteristics: inverted lists are sorted by the document order. At least one list is scanned sequentially. This behavior determines that results are generated in the document order, rather than the order of ranking scores. All the results must be generated in order to return the top K results. Essentially, these algorithms are designed to optimize the semantic pruning, and are incapable of supporting top-K processing.

Top-K processing is not a new problem, and has been extensively studied in other areas, e.g., information retrieval and relational databases. Among the proposed algorithms, the Threshold Algorithm (TA) [14] is the most well-known instance. Given a set of ranked inputs and an aggregation function that aggregates local scores from individual inputs, TA matches results from individual inputs and computes a score threshold for unseen results. Generated results whose scores are greater than the threshold are output without blocking.

A straightforward application of TA in XML keyword search appears in RDIL [5]. It iteratively reads new nodes from one keyword list sorted by the ranking scores of individual

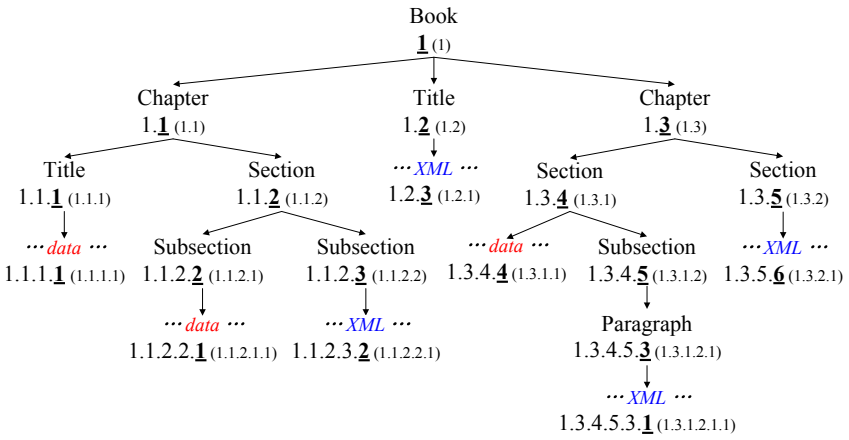


Fig. 1. An Example XML Tree

nodes, and looks up the other inverted lists to generate results. Twig pattern queries with full-text predicates [15] also incorporate a similar idea to support top-K processing. However, for the ELCA/SLCA semantics, all the straightforward applications of TA are not effective. The key problem is the non-trivial semantic pruning which involves complex computations: SLCA prunes the LCAs that are already ancestors of other LCAs; ELCA prunes the LCAs whose keyword occurrences are the descendants of other lower LCAs. In general, XML keyword query evaluation not only computes LCAs of individual nodes, but more importantly needs to check correlations between several LCAs. Applying TA's intuition naively loses the optimization of the semantic pruning and makes it very expensive.

In summary, existing algorithms either (1) focus on the top-K pruning, finding the LCAs that may not be ELCAs or SLCAs and therefore unnecessarily expanding efforts in checking irrelevant nodes, or (2) focus on the semantic pruning, sacrificing the top-K performance. In this paper, we propose a series of join-based algorithms that combine both pruning schemes to support top-K keyword search in XML databases. Specifically, we make the following technical contributions:

1) We propose a join-based algorithm for computing XML keyword query results, which essentially reduces query evaluation to relational joins. The algorithm generates all the results from the lowest level to the highest level, making the semantic pruning very efficient.
2) We incorporate the top-K join from relational databases, and propose a join-based top-K algorithm to compute top K results of XML keyword queries. The algorithm benefits from the efficient pruning of the general join-based algorithm, and incorporates TA's intuition to support top-K processing.
3) We implement the proposed algorithms, perform experiments, and compare with existing approaches. Experimental results demonstrate that our join-based algorithms deliver superior performance not only for the semantic pruning efficiency but also for top-K processing. Moreover, given the mature models and techniques in relational databases, our algorithms are more tractable in real systems.

The rest of the paper is structured as follows: Section II provides preliminaries of XML keyword search, introducing formal definition of the query semantics, ranking functions, and existing algorithms. Section III introduces the join-based algorithm for evaluating XML keyword queries. Section IV introduces the join-based top-K algorithm for the top K results. Experimental results are reported in Section V. Section VI reviews related work, and finally Section VII concludes the paper.

II. Preliminaries

A. Query Semantics

Consider a k-keyword query $\{w_1, \ldots, w_k\}$. Let L_{w_i} be the list of nodes directly containing w_i, and let $lca(v_1, \ldots, v_k)$ be the LCA of nodes $v_1, \ldots, v_k$. The LCAs of the k keywords are defined as $LCA(w_1, \ldots, w_k) = LCA(L_{w_1}, \ldots, L_{w_k}) = \{lca(v_1, \ldots, v_k) | v_1 \in L_{w_1}, \ldots, v_k \in L_{w_k}\}$.

The above definition enumerates all the combinations of nodes in L_{w_1} through L_{w_k}, making the result size exponential to the query size (though many combinations have the same LCA). In the literature, LCA-based variants were proposed, specifying a subset of $LCA(L_{w_1}, \ldots, L_{w_k})$ as the result set. ELCA semantics defines the result as a set of nodes that contain at least one occurrence of all of the query keywords either in their labels or in the labels of their descendant nodes, after *excluding* the occurrences of the keywords in the subtrees that already contain at least one occurrence of all the query keywords. For example, in Figure 1, node 1.1.2 is an answer to the query {XML, data}. However, node 1.1 is not an answer, because its descendant 1.1.2 is already an ELCA, and after excluding the keyword occurrences of 1.1.2, the descendants of 1.1 only contain {data}. SLCA defines a subset of $LCA(L_1, \ldots, L_k)$ such that no LCA in the subset is the ancestor of another LCA. In Figure 1, though 1.1 is the LCA for 1.1.1.1 and 1.1.2.3.2, it is not an SLCA because its descendant 1.1.2 is already an LCA for the two keywords.

The pruning schemes of ELCA and SLCA are non-trivial and algorithmically complex. Given one combination of nodes $v_1 \in L_1, \ldots, v_k \in L_k$ and their LCA $u = lca(v_1, \ldots, v_k)$, whether u is the result or not may be determined by another combination $v_1^{'} \in L_1, \ldots, v_k^{'} \in L_k$: if u is the ancestor of $lca(v_1^{'}, \ldots, v_k^{'})$, then u is not the SLCA; if u is the ancestor of $lca(v_1^{'}, \ldots, v_k^{'})$ and $\exists i$ such that $v_i = v_i^{'}$, then u is not the ELCA. Efficient algorithms must optimize the semantic pruning to avoid not only enumerating all the combinations but also checking the correlations of all the LCAs in the $LCA(L_1, \ldots, L_k)$ space.

B. Ranking Function

XML contains abundant textual contents and structure information. How to rank XML keyword queries incorporating both of them is an interesting problem that has been studied in the literature, e.g. [5], [13], [16], [17]. Since this paper only focuses on the algorithmic perspective, in this subsection, we only introduce one ranking function that is widely adopted in the XML keyword search scenario. Notice that the algorithms we propose in this paper are not restricted to this function.

The basic idea of ranking keyword query results is that individual nodes directly containing the keywords can be viewed as "documents". Local ranking scores are given based on the "documents", and are propagated to their ELCA or SLCA. An aggregation function aggregates them into a global score which is the final ranking score of the result subtree.

Given a node v and a keyword w, $g(v, w)$ is a function that assigns to v a local ranking score. The function g can take multiple factors into account (e.g., IR score that evaluates the content relevance and link-based score that evaluates the global importance of the node), and combine them in an arbitrary way.

Consider the k-keyword query $\{w_1, \ldots, w_k\}$. Let $v^i, i = 1, \ldots, k$, be an occurrence of w_i at depth l^i, and let node u at depth $\widetilde{l}$ be the ELCA/SLCA of v^i's. Let $F(\cdot)$ be the function that combines $g(v^i, w_i)$, $i = 1, \ldots, k$, with *damping factors* into a score of u:

$$score(u) = F(I_1, \ldots, I_k),$$

where $I_i = g(v^i, w_i) \times d(l^i - \widetilde{l})$, $i = 1, \ldots, k$, and $d(\cdot)$ is a decreasing function.

The combining function $F(\cdot)$ is expected to satisfy the following property:

Monotonicity Let u and $u^{'}$ be two ELCAs/SLCAs for the k-keyword query. If $\forall i \in [1, k]$, $I_i \leq I_i^{'}$, then $score(u) \leq score(u^{'})$.

Monotonicity is the assumption on which most top-K processing algorithms are based. It is also true for most existing ranking functions of XML keyword search. For ease of exposition, we simply assume that $F(\cdot)$ is the sum function, i.e., $score(u) = \sum_{i=1}^{k} g(v^i, w_i) \times d(l^i - \widetilde{l})$.

The function $d(\cdot)$ decreases the score of the keyword occurrence as its vertical distance to the ELCA/SLCA increases. It reflects the intuition that compact subtrees are more important because of the tighter relationship between keywords. Similar intuition is also reflected in information retrieval [18]: for the documents that contain all the keywords, if the keyword occurrences in a document are within a short distance, that document's score tends to be higher than the documents whose keyword occurrences are far away.

If the ELCA/SLCA contains more than one occurrences of w_j, i.e., $v_1^j, \ldots, v_m^j$, $F(\cdot)$ only takes the maximum score (after applying the damping factor) of the occurrences as the input, i.e., $\max\{g(v_1^j, w_j) \times d(l_1^j - \widetilde{l}), \ldots, g(v_m^j, w_j) \times d(l_m^j - \widetilde{l})\}$.

C. Existing Algorithms

Many algorithms were proposed to address how to evaluate XML keyword queries efficiently. The basic idea is utilizing the document order of the nodes in the inverted lists to optimize the semantic pruning. Specifically, nodes in the XML tree are identified by Dewey id's. All the Dewey id's are shown within the parentheses in Figure 1. Then computing the LCA of two nodes reduces to computing the longest common prefix of their Dewey id's. The inverted lists $L_1, \ldots, L_k$ are essentially joined to compute the prefixes of the Dewey id's that contain all the keywords.

Two types of algorithms were developed in the literature: stack-based algorithms [5], [10], [6] and index-based algorithms [6], [8], [11]. The stack-based algorithms follow the idea of the merge join in relational databases, and scan the Dewey id's from the smallest to the largest. They use a stack to online merge all the Dewey id's and simultaneously compute the longest prefixes that contain all the keywords. The index-based algorithms, on the other hand, follow the idea of the index join. Given a node v containing one keyword, let u be the LCA for v and its *closest* nodes containing the other keywords. The main observation is that any ELCA/SLCA containing v cannot be lower than u. Thus, the algorithms look up the inverted lists to locate v's closest nodes containing the other keywords, and further compute the ELCA/SLCA.

RDIL in [5] were proposed to support top-K keyword search in XML. It iteratively retrieves new nodes from one inverted list sorted by the local score, and looks up indices of the other inverted lists to generate results. Essentially, it is very similar to the index-based algorithms. However, RDIL has two major problems. First, scanning nodes out of the document order loses the semantic pruning optimization, and has to check many irrelevant LCAs and their correlations. Second, retrieving nodes by the order of the local score may not lead to the results with high overall scores. Given a node containing one keyword, even if its local score is very high, if the prefix of its Dewey id containing the other keywords is very short, the global ranking score will be penalized a lot by the damping factor.

III. Join-based Algorithm

The goal of the join-based algorithms is to incorporate both the semantic pruning and the top-K processing. We concentrate on the efficient semantic pruning in this section. We will further incorporate top-K ideas from relational databases in the next section.

Ancestor-descendant relationships between LCAs are the key for the semantic pruning of the ELCA/SLCA semantics: any LCA must check its descendant LCAs (if there are any). Existing algorithms utilize the order of the Dewey id to efficiently capture ancestor-descendant relationships, and consequently force the processing order to be the document order. However, according to the query semantics, if LCAs at low levels are generated first, LCAs at high levels can be pruned directly based on previously generated LCAs. No document order needs to be enforced. In the following, we will show how to get rid of the document order, meanwhile achieving the efficient semantic pruning.

A. Node Encoding

We first define a new encoding of nodes in the XML tree. Each node in the tree is assigned a number, called *JDewey* number, such that

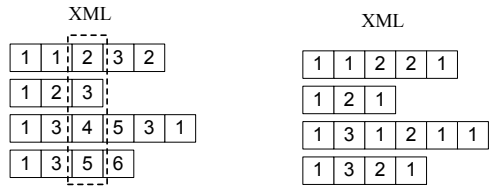
1) the number is a unique identifier among all the nodes in the same tree depth. In Figure 1, the JDewey numbers are underlined under the nodes' tags.
2) for two nodes v_1, v_2 in the same level, if v_1's JDewey number is greater than v_2, all the JDewey numbers of the children of v_1 are greater than the children of v_2. For example, consider two nodes $v_1(1.3.4)$ and $v_2(1.1.2)$ in Figure 1. Nodes v_1 and v_2 are in the same level and v_1's JDewey number (i.e., 4) is greater than v_2 (i.e., 2). So all the JDewey numbers of v_1's children (i.e., 4 and 5) are greater than v_2's children (i.e., 2 and 3).

Given the JDewey numbers of the nodes in the XML tree, the *JDewey sequence* of node v is a vector of JDewey numbers on the path from the root to v. In Figure 1, the JDewey sequences are shown under the tags.

At first glance, the JDewey encoding is very similar to the Dewey id. Their common feature is that ancestor-descendant relationships are implicitly captured in the encoding. The major difference is how they uniquely identify a node in the tree. Let S denote a JDewey sequence and $S(i)$ denote the ith JDewey number in S. Two parameters, i and $S(i)$, can uniquely identify the node, whereas the Dewey id requires the whole vector.

The representation difference may result in different storages of the inverted lists. Figure 2 shows the inverted list of {XML} in both encodings. Since individual numbers in a Dewey id cannot identify nodes, in the inverted list, the whole Dewey id must be encoded into a sequence of bytes and stored integrally. In contrast, JDewey sequences can be broken into individual numbers and the inverted list is stored by column, as shown in Figure 2(a). Here a column corresponds to a level in the XML tree, and numbers in that column identify nodes at that level. In the next subsection, we will see how this column-oriented storage facilitates query evaluation.

With the JDewey sequences, the $lca(\cdot)$ operator no longer needs to match the longest common prefix. Given two nodes v_1, v_2 and their corresponding JDewey sequences S_1, S_2, if i is the largest number such that $S_1(i) = S_2(i) = N$, then node N at level i is the LCA of v_1 and v_2.



(a) JDewey sequences (b) Dewey id's

Fig. 2. Inverted lists in two encodings

An important feature of the JDewey encoding is that numbers in every column of an inverted list are sorted, if the inverted list is ordered by the JDewey sequence, as shown in Figure 2(a). More formally, the order of JDewey sequences is defined as follows: $S_1 < S_2$ iff either (1) $\exists j$, $S_1(j) < S_2(j)$, or (2) S_1 is the prefix of S_2.

Property 3.1: Given two JDewey sequences S_1 and S_2, if $S_1 < S_2$, then $\forall i \leq \min\{|S_1|, |S_2|\}$, $S_1(i) \leq S_2(i)$.

Proof: By the definition of the JDewey order, either (1) S_1 is the prefix of S_2, or (2) $\exists j$, $S_1(j) < S_2(j)$. For the first case, the above property is obviously true. For the second case $S_1(j) < S_2(j)$, since $S_1(j)$ is the parent of $S_1(j+1)$ and $S_2(j)$ is the parent of $S_2(j+1)$, by the second requirement of the JDewey number, $S_1(j+1) < S_2(j+1)$. Similarly, since $S_1(j)$ is the child of $S_1(j-1)$ and $S_2(j)$ is the child of $S_2(j-1)$, $S_1(j-1) \leq S_2(j-1)$ (otherwise, $S_1(j) > S_2(j)$ which contradicts with the condition). By induction, $\forall j \leq \min\{|S_1|, |S_2|\}, S_1(j) \leq S_2(j)$. ■

Now we briefly discuss how to maintain the JDewey encoding. When nodes are removed from the XML tree, their JDewey numbers and the corresponding JDewey sequences are deleted. It is tricky when nodes are inserted into the tree, as the second requirement of the JDewey number must be satisfied. Consider inserting node v as a child of u. The JDewey number assigned to v must be less than the nodes at the same level whose parents' JDewey numbers are greater than u. Also it must be greater than the nodes at the same level whose parents' JDewey numbers are less than u. To address this problem, extra spaces of the JDewey numbers are reserved for u's children. For example, in Figure 1, node 1.1.2 has two children. With 2 extra numbers reserved for 1.1.2, the JDewey number of the children of 1.3.4 is 6, instead of 4. Notice that when all the reserved spaces of u are used, only a partial XML tree needs to be re-encoded. In the example, if node 1.1.2 runs out of space when new nodes are added as its children, only the subtree rooted at 1.1 needs to be re-encoded: update 1.1's JDewey number to be the largest number in the second level, and then corresponding numbers can be chosen for its descendants.

Another concern of this new encoding is that it normally takes more bytes to represent a JDewey sequence than a Dewey id, because the JDewey number requires uniqueness among all the nodes at the same level whereas the Dewey id's number only requires uniqueness among its siblings. In the experimental section, we will show that with storage optimization, the inverted index using the JDewey encoding

is around the same size as existing systems using Dewey id's.

B. Pseudo Algorithm

Now we introduce a join-based algorithm, which reduces keyword query evaluation to relational joins. For simplicity, we focus on the ELCA semantics in the following. We will briefly mention how to evaluate the SLCA semantics later.

Consider two lists of nodes $L_{xml} = \{S_1^1, S_2^1, \ldots, S_m^1\}$ and $L_{data} = \{S_1^2, S_2^2, \ldots, S_n^2\}$ containing {XML} and {data} respectively. Let $l_m^1 = \max\{|S_1^1|, \ldots, |S_m^1|\}$, $l_m^2 = \max\{|S_1^2|, \ldots, |S_n^2|\}$, and $l_m = \min\{l_m^1, l_m^2\}$. For two lists of JDewey numbers, $L_{xml}(l_m) = \{S_1^1(l_m), \ldots, S_m^1(l_m)\}$ and $L_{data}(l_m) = \{S_1^2(l_m), \ldots, S_n^2(l_m)\}$, if a JDewey number N appears in both lists, then node N at level l_m contains all the keywords and is an LCA. In other words, $L_{xml}(l_m) \bowtie L_{data}(l_m)$ computes the LCAs at level l_m. In general, $\forall l \leq l_m$, $L_{xml}(l) \bowtie L_{data}(l)$ computes the LCAs at level l. Notice that for a pair $S_i^1 \in L_{xml}, S_j^2 \in L_{data}$, if $S_i^1(l_m) = S_j^2(l_m)$, then $\forall l < l_m$, $S_i^1(l) = S_j^2(l)$. Thus, if S_i^1 and S_j^2 are joined once, they should not be matched at higher levels.

According to the ELCA semantics, an ELCA should contain at least one occurrence of all the keywords after excluding the keyword occurrences of its descendant ELCAs. The LCAs generated by $L_{xml}(l_m) \bowtie L_{data}(l_m)$ are also ELCAs, because there cannot be any other LCAs lower than l_m. For the matched pair $S_i^1(l_m) = S_j^2(l_m)$, since they are already the occurrences of the generated ELCA and should not be the occurrences of other higher ELCAs, S_i^1 and S_j^2 should be excluded from L_{xml} and L_{data} in the following processing. Similarly, $\forall l < l_m$, if some JDewey numbers are matched through $L_{xml}(l) \bowtie L_{data}(l)$, their corresponding JDewey sequences should be excluded. Then, the LCAs generated by each join are also ELCAs.

Input : $L_{xml} = \{S_1^1, \ldots, S_m^2\}$, $L_{data} = \{S_1^2, \ldots, S_n^2\}$
Output: $R_{\min\{l_m^1, l_m^2\}}, \ldots, R_1$, where R_l is a list of ELCAs at level l

$H_1 \leftarrow \emptyset$, the set of JDewey sequences that have been erased from L_{xml};
$H_2 \leftarrow \emptyset$, the set of JDewey sequences that have been erased from L_{data};
for $l \leftarrow \min\{l_m^1, l_m^2\}$ **to** 1 **do**
  compute relational join $L_{xml}(l) \bowtie L_{data}(l)$;
  foreach *pair* $S_i^1(l) = S_j^2(l)$ **do**
    if $i \notin H_1$ *and* $j \notin H_2$ **then**
      $R_l \leftarrow R_l \cup \{S_i^1(l)\}$;
      $H_1 \leftarrow H_1 \cup \{i\}$;
      $H_2 \leftarrow H_2 \cup \{j\}$;
    end
  end
end
Return $R_l, l = \min\{l_m^1, l_m^2\}, \ldots, 1$ if R_l is not empty ;

Algorithm 1: Join-based algorithm for computing ELCAs

Algorithm 1 shows the pseudo code for computing ELCAs. In each iteration, the algorithm scans the two lists of JDewey numbers and performs the join. The semantic pruning is done by excluding matched JDewey sequences from the following joins. Since the processing is bottom up, the correctness of the semantics is guaranteed. In Section III-A, we mentioned that the inverted lists are stored vertically. Thus, Algorithm 1 is I/O optimized. Moreover, the algorithm does not read the whole JDewey sequences from the disk at once. Note that the scan starts from $l_0 = \min\{l_m^1, l_m^2\}$ (since it is obvious that there is no ELCA lower than l_0). This would save disk I/O when the XML tree is deep and some keywords only appear at high levels.

Example 3.1: Consider the two-keyword query {XML, data}. The inverted lists are shown in Figure 3(a). Since $l_m^1 = 6, l_m^2 = 5$, the join starts from the 5th column, i.e., $\{2, 3\} \bowtie \{1\}$. Since no matched number is found, there is no ELCA at this level. Then move to the next column, as shown in Figure 3(b), and join the next two lists of JDewey numbers, i.e., $\{3, 5, 6\} \bowtie \{1, 2, 4\}$. Again no ELCA is generated. In Figure 3(c), the join between $\{2, 3, 4, 5\}$ and $\{1, 2, 4\}$ finds matched numbers $\{2, 4\}$. So the nodes numbered 2 and 4 at level 3 are the lowest ELCAs. Their corresponding JDewey sequences should also be erased from the following processing, as shown in Figure 3(d) and Figure 3(e). This process repeats until it reaches the root level, and eventually identifies the root as the last ELCA.

It must be explained that in the above example, number 1 appears twice in $L_{xml}(1)$, as shown in Figure 3(e). Two pairs would be matched if we follow the relational join semantics. The two numbers in $L_{xml}(1)$ correspond to the two nodes (1.2.3 and 1.3.5.6) that are both occurrences of {XML}, and the two matched pairs correspond to the same ELCA, i.e., the root. Only one of them needs to be output. In other words, the joins in our scenario follow the *set* semantics, instead of the *bag*.

For the queries with $k > 2$ keywords, the algorithm is the same, except that the initial value of l becomes $\min\{l_m^1, \ldots, l_m^k\}$ and at each level one join becomes $k - 1$ joins.

C. Join Optimization

In this subsection, we discuss join optimizations in three aspects: join algorithms, join ordering, and dynamic optimization.

Join algorithms. By Property 3.1, $\forall l \in [1, \min\{l_m^1, l_m^2\}]$, $L_{xml}(l)$ and $L_{data}(l)$ are sorted. Both the merge join and the index join are available for the join plan. For the index join, since each column is sorted, conceptually no additional indices are required, though in practice sparse indices can be built over columns to improve efficiency.

Having the two join algorithms, the main-memory complexity of Algorithm 1 is given as follows. At each level, $k - 1$ joins are performed, where k is the number of the keywords. For the merge join, the complexity is $O(\sum_{j=1}^{k} |L_j|)$; for the index join, the complexity is $O(k|L_1| \log |L|)$ where $|L_1|$ is the size of the shortest list and $|L|$ is the size of the longest

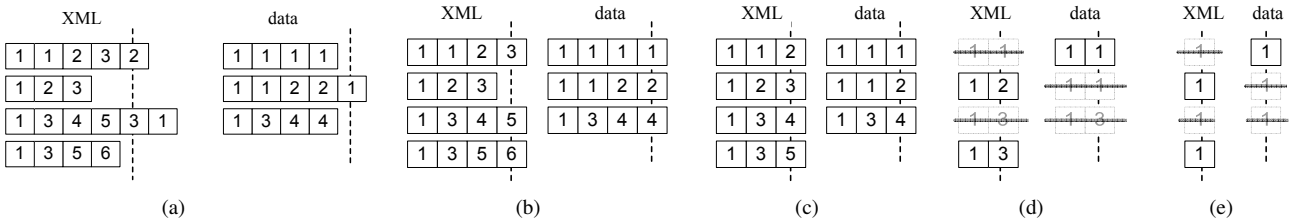


Fig. 3. An execution example of the join-based algorithm for ELCA

list. There are altogether $\min\{l_m^1, \ldots, l_m^k\}$ columns. In the worst case, all the keywords appear in the lowest level. Then $\min\{l_m^1, \ldots, l_m^k\} = d$ where d is the depth of the XML tree. If all the joins use the merge join, the overall complexity would be $O(d \cdot \sum_{j=1}^{k} |L_j|)$; if all the joins use the index join, the overall complexity would be $O(d \cdot k|L_1| \log |L|)$. For comparison, the complexities for the stack-based algorithms and the index-based algorithms are $O(d \sum_{i=1}^{k} |L^i|)$ and $O(dk|L^1| \log |L|)$. The join-based algorithm can leverage existing techniques from relational databases to choose the right join algorithm for queries with various frequencies.

Join ordering. The join order has an important impact on the join performance. A lot of efforts have been made in relational databases. In XML keyword search, the cost model for the join is greatly simplified. First, since all the columns in the inverted lists are already sorted, the sort order of intermediate results is also retained. Second, the join in our scenario follows the set semantics. In general, $|L_{xml}(l) \bowtie L_{data}(l)| \leq \min\{|L_{xml}(l)|, |L_{data}(l)|\}$, whereas in RDBMS, in the worse case, result space can be the Cartesian product of the input relations.

The simplification of the cost model implies a simple yet effective join ordering scheme for XML keyword search. In our system, the join order is always the left-deep join, from the shortest inverted list to the longest inverted list.

Dynamic optimization. Besides query optimization at compile time, the query plan can also be optimized dynamically with accurate knowledge of run-time parameters. In particular, we focus on choosing join algorithms dynamically. Consider a three-keyword query $\{w_1, w_2, w_3\}$ and the join order $(L_{w_1}(l) \bowtie L_{w_2}(l)) \bowtie L_{w_3}(l)$. If the intermediate result size of the first join is orders of magnitude smaller than $L_{w_3}(l)$, the second join would choose the index join; otherwise, the second join would choose the merge join.

More importantly, the join-based algorithm provides an opportunity to choose join algorithms in different contexts. Algorithm 1 processes nodes bottom up, and joins are performed for each column. In the XML tree, different levels may correspond to different contexts and thus have different join selectivities. Consider the DBLP database where papers are organized by conferences. Paper elements contain information about title, authors, etc. Let w_1={topk}, w_2={rewriting} and w_3={XML}, and assume $|L_{w_1}| < |L_{w_2}| < |L_{w_3}|$. In intuition, keyword co-occurrences of {topk} and {rewriting} are few at paper level because very few papers discuss these two topics at the same time. However, at the conference level, their co-occurrences become many because nearly all the database conferences cover these two topics. In general, *keyword correlation* is a concept bound to specific contexts. The selection of the join algorithms should be context-aware. While it is hard to predict keyword correlations in different contexts at compile time, with dynamic optimization, join algorithms can be chosen on the fly. In this example, at the paper level, the second join will normally choose the index join because the result size of the first join is very small. Then at the conference level, it is likely that the second join will switch to the merge join because the result size of the first join may be comparable to $|L_{w_3}|$.

For comparison, the best effort the existing systems can make is choosing the stack-based or the index-based algorithms for individual keywords. Processing the entire Dewey id integrally loses the fine granularity to identify different contexts, and therefore cannot achieve optimality in the execution.

D. Compression

Compression in relational databases improves the performance significantly. Most recently, [19], [20] revisit this topic in the context of column-oriented databases. In XML keyword search, since the inverted lists are stored vertically and all the columns are sorted, the same JDewey numbers are grouped together and stored consecutively, as demonstrated in Figure 3(a). Two compression schemes in [19] are used in our system to improve the performance. For the columns that contain a large number of distinct values, e.g., $L_{xml}(3)$ in Figure 3(a), the first entry of each disk block is the original JDewey number and every subsequent value is the delta from the first JDewey number. For the columns that contain few distinct values, e.g., $L_{xml}(1)$ in Figure 3(a), duplicate JDewey numbers are represented by triples (v, r, c) such that v is a JDewey number, r is the row number where v first appears, and c is the number of times v repeats.

The two compression schemes are very effective in saving storage for the column-oriented inverted lists. The reason the JDewey encoding requires more bytes than the Dewey encoding is the uniqueness requirement of the JDewey number. However, the first scheme only stores delta values within one disk block, achieving a similar effect of the numbering scheme of the Dewey encoding (i.e., relative position among siblings). Furthermore, the second scheme compresses duplicate JDewey numbers into one entry, leveraging the fact that word distribution is usually biased in different contexts. Consider the above DBLP example. While {rewriting} rarely appears in network

conferences, it is a frequent term in database community. Therefore, in the inverted list of {rewriting}, for the column that corresponds to the conference level, the distribution of the JDewey numbers mainly concentrates on a few distinct values each of which corresponds to a database conference. After the compression, same conferences only appear once and the number of triples stored is relatively small. In the traditional Dewey id, although the number assigned to a node v requires less bits for representation, this number has to appear in all the Dewey id's of the descendants of v.

Column-oriented compression also improves the query evaluation efficiency. Recall that duplicate numbers in a column only generate at most one result, as demonstrated in Figure 3(e). The second compression scheme groups the same value in indexing time and saves the online computation.

E. Range Checking

In Algorithm 1, the semantic pruning is done by erasing matched JDewey sequences from the inverted lists. In the compressed columns, a triple corresponds a range that the JDewey number spans. Thus, the semantic pruning can be based on ranges rather than individual rows.

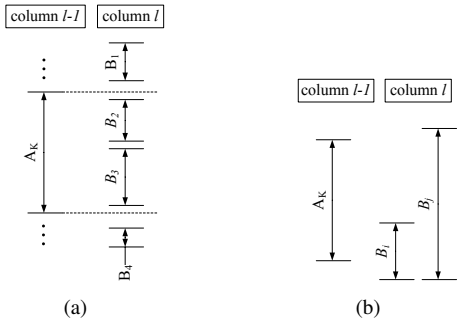


Fig. 4. A snapshot of range checking

Consider a snapshot shown in Figure 4(a), where $B_j, j = 1, \ldots, 4$ are four ranges of the JDewey sequences in $L_{xml}(l)$ that have been matched. A_k is the range of a JDewey number N in column $l-1$ that can join with some number(s) in the other list $L_{data}(l-1)$. According to the semantic pruning, since the JDewey sequences within B_2 and B_3 are the keyword occurrences of other lower ELCAs, they should be excluded from A_k. In other words, if $|A_k| > |B_2|+|B_3|$, N is the ELCA that contains the occurrence(s) of {XML} after the exclusion; otherwise, N is not the ELCA.

Given a range A_k in column $l-1$, we only need to search the ranges within A_k in column l, and check their sizes. Notice that the relationship between the ranges in column l and A_k is either *contained* or *disjoint*. Cases in Figure 4(b) would never happen. This is because the JDewey numbers in B_i or B_j have the same value and thus have the same parent. A_k either contains all of B_i and B_j or neither of them. Having this property, the range checking is simply a binary search process (searching the ranges within A_k). When the join of column $l-1$ finishes, all the sequences within A_k are excluded from L_{xml}.

F. SLCA Evaluation

The SLCA semantics can also be evaluated through Algorithm 1. The only difference is that for two matched JDewey sequences $S_i^1(l) = S_j^2(l)$, the SLCA pruning erases not only S_i^1 from L_{xml}, but also S_k^1 such that $\exists l_0 < l$, $S_k^1(l_0) = S_i^1(l_0)$, since $S_k^1(l_0)$ corresponds to the nodes that are ancestors of $S_i^1(l)$. With regard to the ranking checking, in Figure 4(a), all the numbers within A_k are not SLCAs, regardless of whether $|A_k| > |B_2| + |B_3|$.

IV. Join-based Top-K Algorithm

The efficient semantic pruning proposed in the previous section is the base for top-K keyword search. Without the document order enforcement, it is possible to apply the top-K pruning to find top ranked results first. The join-based algorithm processes nodes bottom up, and individual scores can be updated when propagated upwards. In such a progressive way, the top-K pruning is able to dynamically choose the most promising nodes at each level, which makes the top-K pruning more effective.

The join-based algorithm reduces XML keyword search to relational joins. Intuitively, top-K keyword queries can be evaluated through top-K joins. In this section, we first review the top-K join problem from relational databases, and then propose a join-based top-K algorithm to return the top K results.

A. Review of Top-K Join in RDBMS

The top-K join problem in relational databases has been addressed by [21], [22]. The basic idea of the algorithm is as follows: scan each relation by the descending order of its tuples' ranking scores. Each time a new tuple is retrieved, join this tuple with all the tuples seen from other relations. At any time, a threshold for all the unseen results can be computed. Generated results whose scores are greater than the threshold are output without blocking.

Consider the following SQL query where the three relations are already sorted by the scores, as shown in Figure 5.

SELECT	R_1.id
FROM	R_1, R_2, R_3
WHERE	$R_1.id = R_2.id$ AND $R_2.id = R_3.id$
ORDER BY	$R_1.score + R_2.score + R_3.score$
LIMIT	K

The algorithm maintains a cursor for each relation, and scans the relation by the order of the score. Figure 5 shows a snapshot of the execution. Solid pointers denote cursors' current positions. Three tuples from each relation have been seen so far, and two results are generated. Next time when tuple $(4, 0.5)$ from R_1 is retrieved, the join between $(4, 0.5)$ and the tuples seen from R_2 and R_3 is performed. Newly generated results are put into the result set.

Let s^i denote the score of the next tuple to be retrieved from R_i, and s_m^i denote the maximum score from R_i (in other words, the score of the first tuple of R_i). Then the scores of all the unseen results are bound by $\max\{(s^1 + s_m^2 + s_m^3), (s_m^1 +$

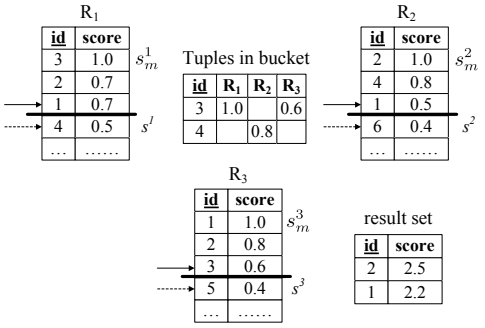


Fig. 5. A snapshot of the relational top-K join

$s^2 + s^3_m), (s^1_m + s^2_m + s^3)\} = \max\{2.5, 2.4, 2.4\} = 2.5$. The score of the result tuple $(2, 2.5)$ is no less than the threshold, and thus can be output, whereas $(1, 2.2)$ is still blocked.

B. Top-K Star Join Algorithm

The top-K join algorithm in the literature is designed for general join patterns. In XML keyword search, the join pattern is only the *star* join, i.e., $R_1.a = R_2.b = R_3.c$, instead of the *sequence* join, i.e., $R_1.a = R_2.b_1$ AND $R_2.b_2 = R_3.c$. Given the property of the star join, there is an opportunity for further improvement. In the following, we propose a new top-K algorithm which computes a tighter upper bound of the unseen results for the star join.

Consider a k-relation star join $R_1.id = R_2.id = \ldots = R_k.id$. The algorithm works as follows: (1) maintain a cursor for each relation, and let s^i be the score of the tuple right after the cursor in R_i. Each time retrieve one tuple t^i from R_i. R_i is chosen in a round-robin way until the result size reaches K. After that, R_i whose s^i is maximum is chosen. (2) Put t^i into the hash bucket. If there is a matched tuple t^0 in the bucket, increment t^0's score by t^i's score. If t^0 has been matched $k-1$ times (there is no match when it is put into the bucket first time), move it from the bucket to the result set.

The threshold of the unseen results for the star join is computed under two cases: (1) results whose id's have not been seen in any relation; (2) results whose id's have been seen in some relation(s), but not all. In other words, their id's reside in the bucket.

- For case 1, their threshold is $\sum_{i=1}^{k} s^i$.
- For case 2, the tuples in the bucket are grouped into groups $G_P, P \subset \{1, \ldots, k\}$. All the tuples in G_P have been seen in $R_j, j \in P$. Let $ms(G_P)$ denote the maximum score of the tuples in G_P. Then the threshold of the tuples in G_P is $ms(G_P) + \sum_{j \notin P} s^j$. The threshold of all the tuples in the bucket is: $\max_{P \subset \{1,\ldots,k\}}(ms(G_P) + \sum_{j \notin P} s^j)$.

Since $ms(G_P) + \sum_{j \notin P} s^j \geq \sum_{i \in P} s^i + \sum_{j \notin P} s^j = \sum_i^m s^i$, we only need to consider the threshold of case 2. Therefore, the threshold of all the unseen results is: $\max_P(ms(G_P) + \sum_{j \notin P} s^j)$, $P \subset \{1, \ldots, k\}$.

For comparison, the threshold of the unseen results for the traditional top-K join algorithm is: $\max_i(s^i + \sum_{j \neq i} s^j_m)$ where $i = 1, \ldots, k$. For any $i \in [1, k]$, let P be the subset of $\{1, \ldots, k\}$ such that $i \notin P$, then $ms(G_P) + \sum_{j \notin P} s^j \leq \sum_{j \in P} s^j_m + \sum_{j \notin P} s^j \leq s^i + \sum_{j \neq i} s^j_m$. Therefore, our algorithm provides a tighter upper bound of the unseen results for the star join. In Figure 5, if we use the new algorithm, two tuples are currently in the bucket, i.e., tuple 3 has been seen in R_1 and R_3, tuple 4 has been seen in R_2, $G_{\{1,3\}} = (3, 1.0 + 0.6) = (3, 1.6)$ and $G_{\{2\}} = (4, 0.8)$, as shown in Figure 5. If they can be results in the future, their scores would be bound by $\max\{1.6 + s^2, 0.8 + s^1 + s^3\} = \max\{2.0, 1.7\} = 2.0$. Thus, the second result $(1, 2.2)$ can also be output without blocking. The tighter upper bound is attributed to the fact that the new algorithm maintains the status of the *partial results*, i.e., the tuples that are *partially joined* within a subset of the relations. To compute the threshold of the partial results, only the scores from the unjoined relations are estimated.

In the algorithm, the number of the groups in the bucket is $2^k - 2$ in the worst case, which seems exponential to the query size (number of relations). However, notice that the number of tuples maintained in the bucket is bound by the number of tuples seen so far. Recall that each time a new tuple is retrieved, both the old and the new algorithms need to generate all the valid join combinations with the tuples seen from other relations. Maintaining the pool of all the tuples already retrieved is the requirement for both algorithms. Grouping within the bucket doesn't increase the algorithm complexity.

C. Join-based Top-K Keyword Search

The idea of the join-based top-K algorithm for keyword search is sorting the inverted lists by the ranking score and using the top-K star join algorithm as the join plan. The joins are performed bottom up and the semantic pruning is achieved by the range checking. However, there exists a problem for such schemes. Consider two nodes v_1, v_2 that directly contain the keyword w. If v_1 is at the level l_1, v_2 is at the level l_2 and $l_1 > l_2$, given the fact that $g(v_1, w) > g(v_2, w)$, the relationship between $g(v_1, w) \times d(l_1 - l_2)$ and $g(v_2, w)$ is unknown before the computation and the comparison. In Figure 6, although the original score of the first JDewey sequence is greater than the second, in the 4th column, the relationship between $0.5 \times d(3)$ and 0.44 may be greater than, equal to, or less than, depending on how fast the original score decreases. This fact means that for L_{xml}, $L_{xml}(l_1)$ and $L_{xml}(l_2)$ may have different orders of JDewey sequences with respect to their ranking scores.

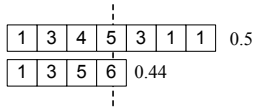


Fig. 6. Two JDewey Sequences with ranking scores

To overcome this problem, JDewey sequences in L_{xml} are grouped by their lengths, as shown in Figure 7. Within one group, $\forall S_1, S_2$, if $score(S_1(l)) > score(S_2(l))$, $score(S_1(l - l_0)) > score(S_2(l - l_0))$. In other words, there is an unique

order for JDewey sequences in one group. The number of the groups is at most the height of the XML tree. This scheme breaks L_{xml} into segments each of which is ordered by the local ranking scores. The complete order of a column can be reconstructed by merging segments online. In the implementation, the algorithm maintains a cursor for each segment. Recall that the top-K join algorithm only retrieves one JDewey number from the column at one time. So the algorithm picks one JDewey number with the highest score from all the cursors at each iteration.

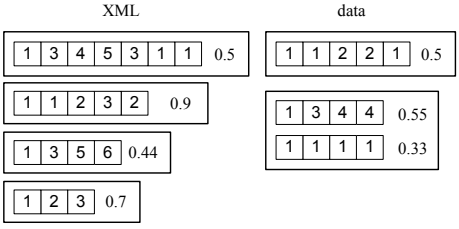


Fig. 7. JDewey sequences grouped by their lengths

Similar to the general join-based algorithm, the join-based top-K algorithm joins JDewey numbers by column. The only difference is that the top-K star join algorithm is used as the join plan. The generated results whose ranking scores are greater than the threshold of the unseen results are output without blocking. However, notice that the algorithm in Section IV-B is for one join and only computes the upper bound of the unseen results within the current column. Since ELCAs may be generated by all the columns, we also need to compute the threshold of the unseen results in other columns. More precisely, if we are currently performing the join of column l_0, $\forall l < l_0$, we also need to compute the upper bound of ELCAs at level l. The upper bound of the ELCAs at level l can be computed as $\sum_{i=1}^{k} s_m^i(l)$ where $s_m^i(l)$ is the maximum score in $L_i(l)$ and k is the number of keywords. In practice, we do not need to compute all the columns. Instead, if (1) $l < l_0 - 1$ and (2) $\forall i \in [1,k]$, $\nexists S \in L_i$ such that $|S| = l$, then we can skip column l. This is because: if the above two conditions are both true, $\forall i \in [1,k]$, $s_m^i(l) = s_m^i(l+1) \times d(\cdot) < s_m^i(l+1)$. Therefore, the threshold of the unseen results in column l is always less than the unseen results in column $l+1$. On the other hand, if $\exists S \in L_i$ such that $|S| = l$, there may be no damping factor for $s_m^i(l)$ and thus the upper bound of this column must be computed.

Example 4.1: Consider again the query {XML, data}. Figure 7 shows the two inverted lists and the original ranking scores of JDewey sequences. Assume the damping function is $d(\Delta l) = 0.9^{\Delta l}$. The joins for column 5 and 4 are first performed, and no result is generated. Figure 8(a) shows the status of the join for column 3. In the figure, two numbers from $L_{xml}(3)$ (i.e., $2, 3$) and $L_{data}(3)$ (i.e., $2, 4$) have been retrieved and put into the hash bucket. Number 2 is matched and further moved into the result set. Its score is $0.73 + 0.41 = 1.14$. The threshold of the unseen results in column 3 is: $\max\{ms(G_{\{1\}}) + s^2(3), ms(G_{\{2\}}) + s^1(3)\} = \max\{0.7 + 0.3, 0.5 + 0.4\} = 1$. We also need to consider the unseen results in other columns, i.e., column 1 and column 2. Since both $L_{xml}(1)$ and $L_{data}(1)$ do not contain sequence S such that $|S| = 1$, we only need to consider column 2. The maximum scores from $L_{xml}(2)$ and $L_{data}(2)$ are $0.7 * 0.9 = 0.63$ and $0.5 * 0.9 = 0.45$. So the threshold of the unseen results in column 2 is $0.63 + 0.45 = 1.08$, which is also less than the node 2's score. Therefore, node 2 at level 3 can be output.

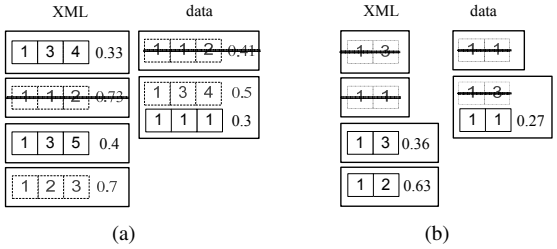


Fig. 8. A snapshot of the join-based top-K algorithm

When all the numbers in $L_{xml}(3)$ and $L_{data}(3)$ are retrieved, number 4 in column 3 is also matched and its score is $0.33 + 0.5 = 0.88$. However, it cannot be output at this point. As shown in Figure 8(b), the upper bound of the unseen results in column 2 is $0.63 + 0.27 = 0.9$, which is greater than the node 4's score.

V. Experiments

In this section, we experimentally evaluate the join-based algorithms on DBLP and XMark data sets, and compare them with the stack-based [5] and the index-based [8] systems. The size of the DBLP document is 496MB. XMark is generated with factor 1.0 and the size of the document is 113MB. Considering the original DBLP XML tree is very shallow, we group the papers firstly by conference/journal names, and then by years. Xcerse and Lucene are used to parse the XML tree and the textual contents. All the algorithms are implemented using Java under JDK 5. Instead of using column-oriented databases, we store the inverted lists directly on the disk. The main reason is that the number of the keywords is very large (more than 300,000 in DBLP) and most inverted lists are very short. We also build sparse indices on the columns to improve the efficiency of the index join. All the experiments are performed on a Debian 2.40GHz PC with 1G memory. Similar to the previous technical discussion, we only focus on the ELCA semantics in the following. Query execution time for the SLCA semantics is around the same as the ELCA semantics for any algorithm.

A. Index Size

Table I shows the index sizes of different algorithms, where *IL* denotes the inverted lists and *sparse* denotes the sparse indices. As we can see, the JDewey encoding on which the join-based algorithms rely does not introduce much space overhead (and even saves spaces for the DBLP data set). This is mainly due to the effectiveness of the compression,

TABLE I
INDEX SIZES OF DIFFERENT ALGORITHMS

	DBLP		XMark	
Join-based	IL	sparse	IL	sparse
	327MB	14MB	302MB	4MB
stack-based	392MB		267MB	
index-based	2.1G		1.3G	
Top-K Join	IL	sparse	IL	sparse
	394MB	14MB	351MB	4MB
RDIL	IL	B+-tree	IL	B+-tree
	392MB	446MB	267MB	252MB

as discussed in Section III-D. In the experiment, Dewey id's are also compressed by the coding scheme proposed in [6].

The index size of the index-based algorithm is extremely large. This is because the implementation in [6], [8] uses a single B-tree in BerkeleyDB. Each key entry in the B-tree is a pair of a keyword and a Dewey Id. If the length of the inverted list for a keyword is n, then this keyword occurs n times in the B-tree, which costs a lot of space.

The lower half of Table I shows the index sizes of the two top-K algorithms. In the top-K scenario, our algorithm has a great advantage in terms of the index size. RDIL, as mentioned in Section II-C, builds additional B-trees on top of the inverted lists, which inevitably introduces much space overhead. On the other hand, the core operation of the join-based top-K algorithm is the hash join, and thus requires no additional index.

B. Query Performance for the Complete Result Set

We compare the query performance of the three algorithms for the ELCA semantics, varying both keyword frequencies and the number of keywords. For each experiment, forty queries within each frequency range are randomly selected. The execution time in the figures is the average of the forty queries executed 5 times. Furthermore, all the experiments are on hot cache. For the index-based algorithm in [8], BerkeleyDB provides an application-level cache mechanism. The stack-based and the join-based algorithm use the cache provided by the file system. Note that the sparse indices are fairly small, as shown in Table I, and are always cached in main memory.

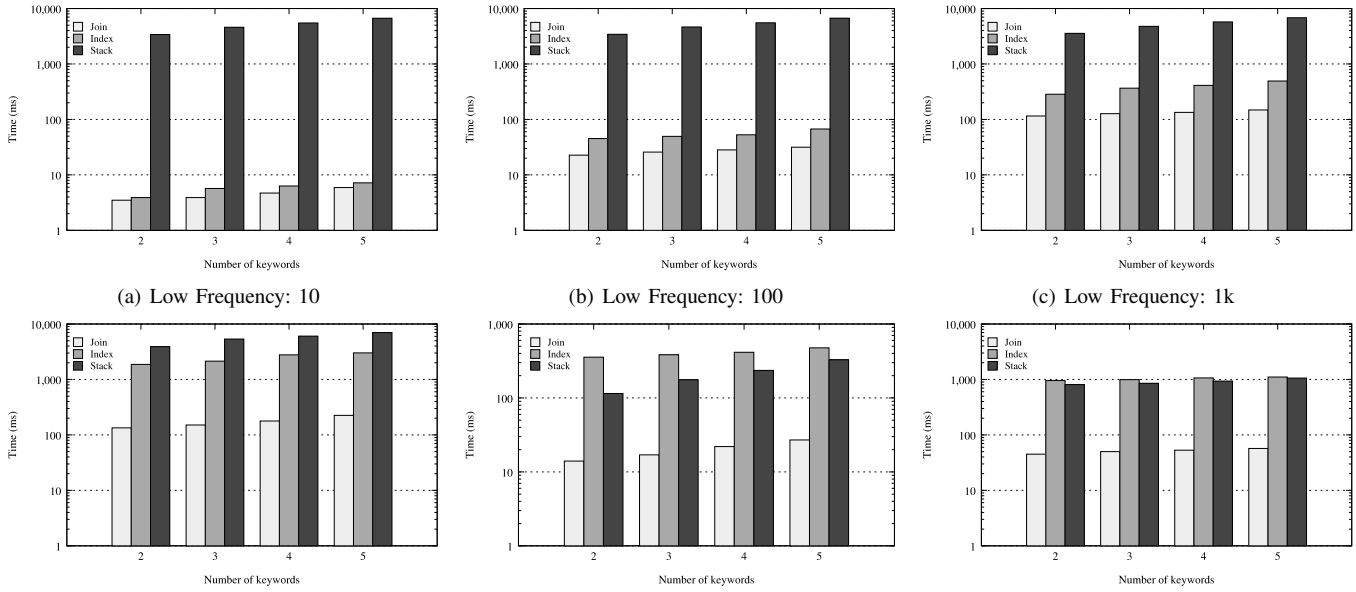
Experimental results are shown in Figures 9(a) – 9(d). Due to the space limit, only results from DBLP are listed. Results from XMark are similar. In fact, query execution time mainly depends on two factors: keyword frequencies and keyword correlations. We vary the number of keywords from 2 to 5. In all queries, the high frequency is fixed, i.e., 100k. The low frequency varies from 10 to 10k. As we can see from the figure, when the low frequency is extremely small (10 or 100), the execution time of our algorithm and the index-based algorithm is in the same order of magnitude. However, when the low frequency goes beyond 1000, the difference is obvious, especially in Figure 9(d). For the queries in that range, our algorithm already switches to the merge join. In fact, if we force the query plan to use the index join, the performance can be as bad as the index-based algorithm. The execution time of the stack-based algorithm is always in the same order of magnitude, regardless of the low frequency. This is because the stack-based algorithms need to scan all the input lists. In consequence, its execution time is bound by the keyword with the highest frequency, which is fixed in the experiment.

We also evaluate the algorithms on keywords with the same frequency, as shown in Figures 9(e) – 9(f). The stack-based algorithm then performs slightly better than the index-based algorithm, which can also be seen from their theoretical complexities. In these experiments, the join-based algorithm performs much better than the stack-based algorithm. This is attributed to several reasons: first, the dynamic optimization chooses the join algorithms based on the size of intermediate results and may not stick to the merge join, though the input inverted lists are around the same size. Since the correlations between randomly selected keywords are normally low, only the first join at each level chooses the merge join. Second, the stack-based algorithms push Dewey id's from all the inverted lists into one stack. When pushing the Dewey id's from the same inverted list into the stack, the algorithm matches their common prefixes and groups them online. For the join-based algorithm, this process is actually done by the second compression scheme in the indexing time which saves the online computation.

C. Query Performance for the Top K Results

Now we evaluate the performance for the top K results and compare three algorithms: the join-based top-K algorithm, the general join-based algorithm that generates the complete result set and RDIL. We first run the algorithms on queries randomly selected in the previous experiments. The result is shown in Figure 10(a). Overall, the performance of the join-based top-K algorithm is worse than the general join-based algorithm. When the low frequency is very small (10 or 100), it is even worse than RDIL. Interestingly, the execution time of the join-based top-K algorithm decreases as the low frequency increases. All these observations are attributed to the keyword correlations. Conceptually, the join-based top-K algorithm only performs well when the number of results is fairly large. For the keywords with low correlations, the total number of results is very small, and the algorithm ends up with scanning all the input lists. Since the high frequency is fixed in the experiment, the execution time is very large. On the other hand, RDIL is similar to the index-based algorithms and can terminate when the shortest list is completely scanned. When the low frequency increases, the number of results also increases. Therefore, it takes less time for the join-based top-K algorithm to find the top 10 results.

While randomly selected queries have low correlations, we manually pick a set of queries such as {sensor, network} and {XML, keyword, search}, and run the three algorithms again. The results are reported in Figures 10(b) and 10(c). Overall, the join-based top-K algorithm is efficient. For most queries,



(a) Low Frequency: 10 (b) Low Frequency: 100 (c) Low Frequency: 1k

(d) Low Frequency: 10k (e) Same Frequency: 1k (f) Same Frequency: 10k

Fig. 9. Query performance for the complete result set

the algorithm terminates much earlier than the general join-based algorithm. This is very important in practice, because the keywords input by users normally have medium or high correlations. RDIL, on the other hand, is much less effective in terms of top-K processing. The reasons were analyzed in Section II-C.

The results in Figure 10 imply that the join-based top-K algorithm and the join-based algorithm for complete results are complementary to each other. The factor that determines their relative performance is the keyword correlation, which is also know as join cardinality in the relational join scenario.

D. Discussion on Hybrid Index

Since the algorithms for complete results and the top K results have strengths in different directions, it is straightforward to design a hybrid index where score indices are built on top of the inverted lists that are sorted by the JDewey sequence. While this approach increases the index size, it makes the three join plans (the merge join, the index join, the top-K join) all available. Whether to choose the top-K join or not will be mainly based on join cardinality estimation: the top-K algorithm should only be used at the current level when the result size is estimated to be large. Moreover, the join algorithms are chosen based on columns and join cardinality is re-estimated for different contexts.

Note that join cardinality estimation is a well-defined problem that has been widely studied in the context of relational databases. A hybrid algorithm is also proposed in [5], combining the stacked-based algorithm with RDIL. However, its cost model is unclear, which greatly limits its practical usage.

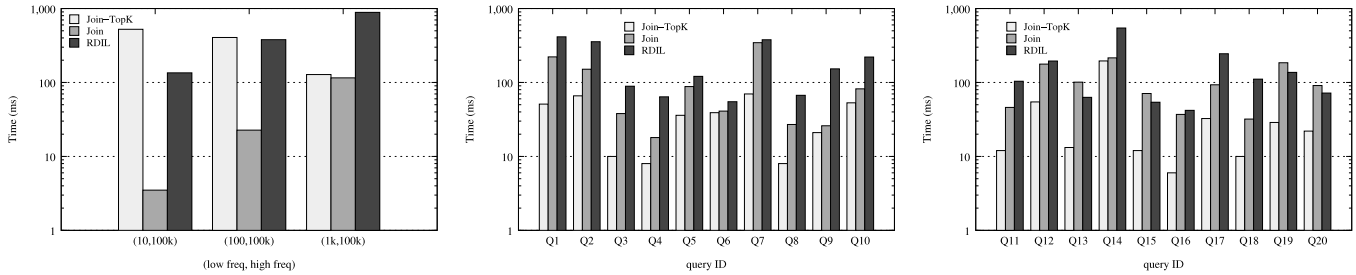
VI. Related Work

Extensive work has been done in LCA-based XML keyword search [5], [6], [8], [10], [11], [9]. Most recently, much work has been done in new problems in this area. For example, [23] studies query evaluation over virtual views of XML. The major difference with the LCA-based keyword search is that given the view definition, the returned elements are fixed, whereas the returned results of the LCA-based semantics can be arbitrary elements in XML tree. [10], [12] study the problem of how to return results with more semantics. They firstly compute SLCAs using the index-based algorithm, and then further infer relevant results by analyzing matched patterns and XML structures.

Another set of work, e.g. [16], [17], [15], [9], [13], tries to extend XQuery with the full-text predicates, combining both the IR ranking mechanism and the XML tree structure to improve search effectiveness. Keyword proximity search in XML [7], [24] shares many similarities with LCA-based keyword search. However, the semantic pruning is not considered in their scenarios.

In addition to semi-structured data, there is also much work on keyword search over structured data. DISCOVER [3], DBXplorer [1] and BANKS [2] are the first three systems presented to support keyword search in relational databases. Their query semantics is that results of keyword queries are sets of tuples that contain all the keywords and can be connected through primary keys and foreign keys. Later work follows this semantics and further focuses on two aspects: efficiency [3] and effectiveness [4], [25]. The top-K processing issue is also studied [22], [25].

Top-K queries in relational databases have been studied for a couple of years. Existing work attacks the problem from different dimensions: monotonic ranking functions [14], [26], [27], [28], non-monotonic ranking functions [29], [25], existence of materialized views [30], [31], [32]. More related work to our scenario is the top-K join problem [21], [22],



(a) Randomly selected 2-keyword queries (b) First set of manually selected queries (c) Second set of manually selected queries

Fig. 10. Query performance for top 10 results

which considers the traditional SQL join semantics. In this paper, we convert keyword search into relational joins. There exists a good possibility to exploit more top-K processing techniques from relational databases and apply them in XML keyword search.

VII. Conclusion

Top-K keyword search in XML is an important issue that has yet received very little attention. Existing algorithms either focus on efficiency, generating results in the document order rather than the ranking order, or simply apply the top-K intuition from other areas, making the query evaluation very expensive. In this paper, we proposed a series of algorithms that incorporate both the efficient semantic pruning and the top-K processing to support top-K keyword search. We presented a join-based algorithm that processes nodes bottom up and reduces keyword query evaluation into relational joins. Several optimizations were proposed to further improve its efficiency. Then we incorporated the idea of the top-K join from relational databases and proposed a join-based top-K algorithm to compute top K results. Extensive experimental results confirmed the advantages of our algorithms over previous algorithms in both efficiency and top-K processing.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A system for keyword-based search over relational databases," in *ICDE*, 2002, pp. 5–16.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *ICDE*, 2002, pp. 431–440.

[3] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *VLDB*, 2002, pp. 670–681.

[4] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury, "Effective keyword search in relational databases," in *SIGMOD Conference*, 2006, pp. 563–574.

[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," in *SIGMOD Conference*, 2003, pp. 16–27.

[6] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest lcas in xml databases," in *SIGMOD Conference*, 2005, pp. 537–538.

[7] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword proximity search in xml trees," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 4, pp. 525–539, 2006.

[8] Y. Xu and Y. Papakonstantinou, "Efficient lca based keyword search in xml data," in *EDBT*, 2008, pp. 535–546.

[9] Y. Li, C. Yu, and H. V. Jagadish, "Schema-free xquery," in *VLDB*, 2004, pp. 72–83.

[10] Z. Liu and Y. Chen, "Identifying meaningful return information for xml keyword search," in *SIGMOD Conference*, 2007, pp. 329–340.

[11] C. Sun, C. Y. Chan, and A. K. Goenka, "Multiway slca-based keyword search in xml data," in *WWW*, 2007, pp. 1043–1052.

[12] Z. Liu and Y. Chen, "Reasoning and identifying relevant matches for xml keyword search," *PVLDB*, vol. 1, no. 1, pp. 921–932, 2008.

[13] S. Amer-Yahia and M. Lalmas, "Xml search: languages, inex and scoring," *SIGMOD Record*, vol. 35, no. 4, pp. 16–23, 2006.

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

[15] M. Theobald, R. Schenkel, and G. Weikum, "An efficient and versatile query engine for topx search," in *VLDB*, 2005, pp. 625–636.

[16] S. Amer-Yahia, E. Curtmola, and A. Deutsch, "Flexible and efficient xml search with complex full-text predicates," in *SIGMOD Conference*, 2006, pp. 575–586.

[17] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "Texquery: a full-text search extension to xquery," in *WWW*, 2004, pp. 583–594.

[18] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[19] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented dbms," in *VLDB*, 2005, pp. 553–564.

[20] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD Conference*, 2006, pp. 671–682.

[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid, "Supporting top-k join queries in relational databases," in *VLDB*, 2003, pp. 754–765.

[22] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient ir-style keyword search over relational databases," in *VLDB*, 2003, pp. 850–861.

[23] F. Shao, L. Guo, C. Botev, A. Bhaskar, M. M. M. Chettiar, F. Y. 0002, and J. Shanmugasundaram, "Efficient keyword search over virtual xml views," in *VLDB*, 2007, pp. 1057–1068.

[24] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on xml graphs," in *ICDE*, 2003, pp. 367–378.

[25] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: top-k keyword query in relational databases," in *SIGMOD Conference*, 2007, pp. 115–126.

[26] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002, pp. 346–357.

[27] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum, "Io-top-k: Index-access optimized top-k query processing," in *VLDB*, 2006, pp. 475–486.

[28] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung, "Efficient aggregation of ranked inputs," in *ICDE*, 2006, p. 72.

[29] D. Xin, J. Han, and K. C.-C. Chang, "Progressive and selective merge: computing top-k with ad-hoc ranking functions," in *SIGMOD Conference*, 2007, pp. 103–114.

[30] V. Hristidis, N. Koudas, and Y. Papakonstantinou, "Prefer: A system for the efficient execution of multi-parametric ranked queries," in *SIGMOD Conference*, 2001, pp. 259–270.

[31] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views," in *VLDB*, 2006, pp. 451–462.

[32] N. Bansal, S. Guha, and N. Koudas, "Ad-hoc aggregations of ranked lists in the presence of hierarchies," in *SIGMOD Conference*, 2008, pp. 67–78.