Join-Based Algorithms for Keyword Search in XML Databases

Liang Chen
University of California, San Diego
jeffchen@cs.ucsd.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

ABSTRACT

We consider the problem of keyword search in XML databases under the *excluding lowest common ancestor* (ELCA) semantics. Our analysis shows that ELCA semantics may lead to conflict with keyword proximity concept, and under such semantics, lower ELCAs are preferable because lower elements tend to be more specific. However, existing algorithms (stack-based and index-based) do not provide efficient support either for this *the lower the better* intuition or for general ranking functions, which is mainly due to the fact that generated results follow the document order. In this paper, we propose a join-based algorithm to compute complete ELCAs, which achieves complexity optimality for queries with various frequencies, as well as guarantees that lowest ELCAs are generated first. More importantly, we shed the new light on the connection between relational join and XML keyword search. Basically, many mature techniques in relational databases can be leveraged in this scenario to optimize query plan and improve execution efficiency. We further adopt the idea from top-K join in relational databases and propose a top-K algorithm for one type of ranking functions. Extensive experimental results demonstrate that the proposed algorithms outperform existing systems.

1. INTRODUCTION

Keyword search has been proven an effective information discovery method for unstructured data (e.g. textual documents), semi-structured data (e.g. XML databases), and structured data (e.g. relational databases). For semi-structured and structured data, keyword search allows users without prior knowledge of schema and query languages (e.g. SQL for relational databases and XQuery for XML databases) to exploit the data. The high level intuition of query semantics of keyword search in semi-structured and structured data is that keywords can spread over multiple elements or tuples in the databases. A result of the query is a set of elements or tuples that contain all the keywords and "connect" with each other in some way. In XML databases, because of the hierarchical structure of XML tree, the connection is captured by the common ancestor of elements. Figure 1 shows a fragment of a XML tree. For the query {XML, data}, node 1.1.2.2.1 and node 1.1.2.3.2 contain the two keywords separately, and node 1.1.2 is the ancestor connecting them. So node 1.1.2 is expected to be the result.

LCA semantics has become prevalent in XML keyword search scenario. Serval previous work has proposed different semantic variances, e.g. SLCA, ELCA, MLCA, VLCA, and efficient algorithms to compute results [11, 28, 29, 25, 18]. In this paper, we focus on the ELCA semantics. We analyze the semantics and existing algorithms, showing that ELCA semantics may lead to conflict with keyword proximity concept, and all existing algorithms do not provide efficient top K support for general ranking purposes.

1.1 Semantics Analysis

According to the ELCA semantics originally proposed in [11], the result of keyword query is 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 query {XML, data}. However, node 1.1 is not the answer, because its descendant 1.1.2 is already an ELCA. After excluding the subtree rooted at 1.1.2, node 1.1's descendants only contain keyword {data}.

Generally, common ancestors reflect how close keywords are connected in the XML tree. However, the ELCA semantics may lead to conflict with proximity concept. Consider three nodes in Figure 1, v_1(1.3.4.4), v_2(1.3.4.5.3.1.1) and v_3(1.3.5.6). By intuition, v_1 and v_3 is closer than v_1 and v_2, because v_2 needs to trace more steps upward to meet with v_1. This statement is also true for existing proximity measures, e.g. number of edges in the subtree that connects all the keywords. However, if we follow the ELCA semantics, 1.3 is no longer the result, because its descendant 1.3.4 is already an ELCA.

The above example reveals that ELCA semantics has some inherent differences with proximity concept, which can lead to different results. The intuition behind the ELCA semantics implies the *the lower the better* rule: in the hierarchical tree structure, people tend to believe that lower elements contain more specific information, and thus are more important than higher elements. In the above example, v_1 and v_2 are in one section which is more specific than chapter. On the other hand, proximity is another measure reflecting

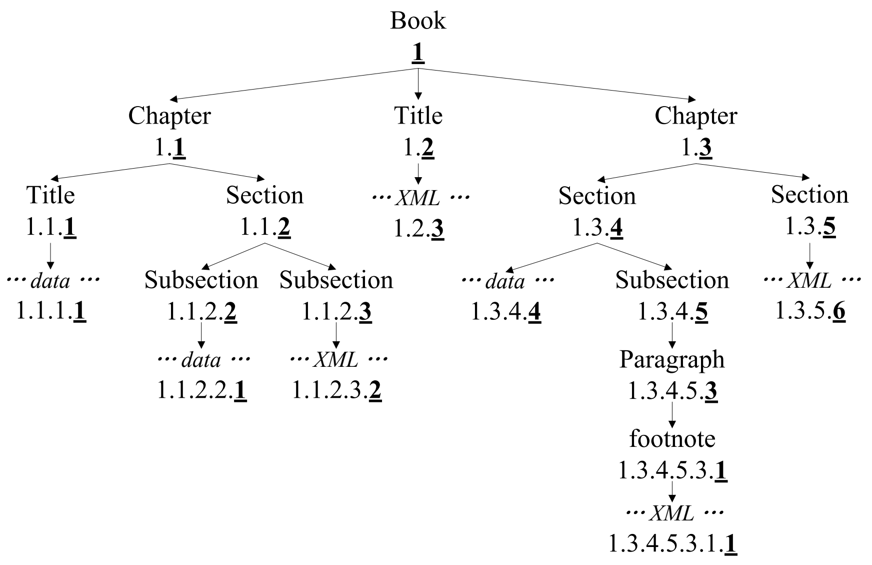


Figure 1: Example XML Tree

different intuition. Since v_1 and v_2 are two sections under one chapter that directly contain the keywords, people may think they are a good summarization of this chapter and thus the whole chapter should be returned. In contrast, v_3 is only a footnote and may have nothing to do with the content of this section.

Therefore, we argue that *the lower the better* rule and *proximity* rule reflect different ranking purposes in XML databases. Under the ELCA semantics, people tend to prefer results at lower level. In the next subsection, we analyze existing algorithms for ELCAs, showing that none of them provide efficient support to return top results, either for *the lower the better* rule or for general ranking functions.

1.2 Algorithms Analysis

Many algorithms have been proposed to efficiently compute ELCAs and its variances. Most of them reply on the following two ideas:

1. Nodes in the XML tree can be represented by Dewey Id, e.g. 1.1.1 of Title node under chapter 1.1 in Figure 1. Then computing LCA of two nodes becomes computing the longest common prefix of two Dewey Ids. Specifically, stack is used to process all the nodes in the document order. Nodes are pushed into the stack one by one. When some node v is to be pushed, all the nodes in the stack that are not ancestors of v are popped out. Popped nodes containing all the keywords are identified as ELCAs. In Figure 1, when node 1.2.3 is to be pushed into the stack, all the nodes under chapter 1.1 are popped out. And node 1.1.2 is outputted as a result.

2. The second idea notices the fact that given a node v containing one keyword, it is very likely that the common ancestor for v and its closest nodes (in document order) containing other keywords is also an ELCA. In Figure 1, node 1.1.2.2.1 contains {data}, and node 1.1.2.3.2 is its closest node containing {XML}. Their common ancestor is also an ELCA for the query. Following this direction, for nodes containing one keyword, index lookup (binary search) is used to locate their closest nodes containing other keywords, and further computes ELCAs. Since binary search is fast and does not need to scan all the nodes containing other words, this type of algorithms can be very efficient if some keyword's frequency is much lower than others.

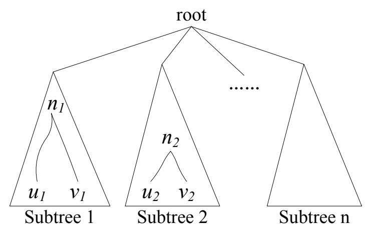


Figure 2: Processing Order of Existing Algorithms for ELCA

In implementation, these two ideas share some common characteristics: nodes for each keyword are sorted by their encodings, and at least one list are scanned sequentially. This behavior determines that ELCAs are generated in the document order, rather than the ranking order. In Figure 2, ELCA n_1 in subtree 1 is generated first, and then n_2 in subtree 2, and so on. This fact means that all the existing algorithms replying on these ideas cannot provide an efficient support for top-K processing, no matter what the actual ranking function is. In Figure 2, n_2 is expected to be better than n_1, either for *the lower the better* rule or the *proximity* rule. However, there is no way to generate n_2 first. Generally, we have to wait until all the ELCAs are generated in order to return top K results.

RDIL in [11] is the first algorithm proposed to return the top K results according to its ranking function. Each time it reads a new node from one list sorted by the ranking score, and looks up indices of other lists to generate ELCAs. Essentially, it is very similar to index-based algorithm. The major problem is that individual nodes with higher ranking scores may not lead to results with higher overall ranking scores.

It also must be explained that although our analysis focuses on the ELCA, it is also applied to work for other semantics, e.g. SLCA: the order of generated results in those systems also follows the document order.

1.3 Contribution

In this paper, we propose a series of join-based algorithms to return ELCAs, either following *the lower the better* rule (generating lowest ELCAs firstly) or according to one type of ranking functions. More importantly, we shed light on the connection between XML keyword search and relational join. Many mature techniques in the relational area, e.g. join ordering, join cardinality estimation, can be leveraged in this scenario. Specifically, we make the following technical contributions:

1. We analyze the semantic conflict between ELCA and keyword proximity, and identify *the lower the better* rule and *proximity* rule for different ranking purposes in XML keyword search.

2. We propose a join-based algorithm to compute complete ELCAs. Similar to relational join, join-based algorithm dynamically choose join plan and thus can achieve performance optimality for various queries. Meanwhile, the algorithm guarantees that lowest ELCAs are generated first. We also propose several optimizations to further improve the efficiency.

3. We adopt ideas from top-K join in relational databases, and propose a new join-based algorithm to compute top-K ELCAs according to one type of ranking functions. The algorithm scans all the lists by the descending order of their ranking scores, and incrementally generates ELCAs. At any time, upper bound of unseen results can be estimated, and those results greater than the bound can be outputted without blocking.

4. We implement proposed algorithms and perform extensive experiment. Comparisons with other approaches validate our superiority. Moreover, given the mature models and techniques in relational join literature, our algorithms are more tractable in real systems.

The following of the paper is organized as follows: Section 2 defines a new encoding of nodes in XML tree which join-based algorithms reply on. Section 3 introduces the algorithm for computing complete ELCAs and its implementation details for the performance purpose. Section 4 discusses the details of the top-K algorithm for one type of ranking functions. Experimental results are reported in Section 5. Section 6 reviews related work. Section 7 concludes the paper with high level comparison and summary.

2. NODE ENCODING

In this section, we define a new encoding for nodes in XML tree. Each node in the tree is assigned a number, called *JDewey Number*, such that

1. the number is an unique identifier among all the nodes in the same level. In Figure 1, JDewey Numbers are underlined numbers under node's tag.

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. Consider two nodes $v_1(1.3.4)$ and $v_2(1.1.2)$ in Figure 1. v_1 and v_2 are in the same level and v_1's JDewey Number (4) is greater than v_2 (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 JDewey Numbers of all the nodes in XML tree, a *JDewey Sequence* of node v is a path vector of JDewsey Numbers from the root to v. In Figure 1, all the JDewey Sequences are shown under node tags.

Although JDewey Sequence is very similar to Dewey Id, there is an important difference between them with respect to computing common ancestors. Let S denote a JDewey Sequence, $S(i)$ denote the ith JDewey Number in S. Given two nodes v_1, v_2 and their JDewey Sequences S_1, S_2, if $S_1(i) = S_2(i) = N$, node N at the ith level is the common ancestor of v_1 and v_2. Here, we only need to compare the ith JDewey Numbers in S_1 and S_2 to identify the ancestor, without considering the prefix of the first $i-1$ numbers. This is because only two parameters, JDewey Number and depth, are needed to identify a unique node in the tree.

The order of two JDewey Sequences is defined as below:

1. $S_1 = S_2$ iff $|S_1| = |S_2|$ and $\forall i \leq |S_1|, S_1(i) = S_2(i)$, where $|S|$ denotes the length of S.

2. $S_1 < S_2$ iff either (1) $\exists j$, $S_1(j) < S_2(j)$, and $\forall i < j$, $S_1(i) = S_2(i)$, or (2) S_1 is the prefix of S_2.

For example, for two JDewey Sequences $S_1 = 1.2.3$ and $S_2 = 1.3.5.6$ in Figure 1, $S_1(2) = 2 < S_2(2) = 3$ and $\forall i < 2$ (i.e. $i = 1$), $S_1(i) = S_2(i) = 1$. So the order of the two JDewey Sequences is: $S_1 < S_2$.

PROPERTY 2.1. *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 order, S_1 is either the prefix of S_2, or $\exists j$, $S_1(j) < S_2(j)$ and $\forall i < j, S_1(i) = S_2(i)$. For the first case, the above property is obvious true. For the second case $S_1(j) < S_2(j)$, since $S_1(j)$ is the parent of $S_1(j+1)$, $S_2(j)$ is the parent of $S_2(j+1)$, by the second requirement of JDewey Number, $S_1(j+1) < S_2(j+1)$. By induction, $\forall (j+k) \leq min\{|S_1|, |S_2|\}, S_1(j+k) < S_2(j+k)$. Therefore, $\forall i < min\{|S_1|, |S_2|\}$, $S_1(i) \leq S_2(i)$. □

For multiple XML documents, document id is attached to the head of JDewey Sequences, and uniqueness of JDewey Number at one level is only required within each document. For JDewey Sequences from different documents, the above property does not hold. However, it has no influence on the keyword search, because we do not need to consider ELCAs across documents and algorithm is only applied on JDewey Sequences within individual documents.

One concern of this encoding is that it normally takes more bytes to represent a JDewey Sequence than traditional Dewey Id, because JDewey Number requires uniqueness among all the nodes at the same level whereas in Dewey Id number assigned to a node only requires uniqueness among its siblings. However, in the experiment part, we will show that index based on JDewey Sequence is around the same size as existing systems.

3. JOIN-BASED ALGORITHM FOR COMPLETE ELCAS

As mentioned in introduction, ELCA semantics implies *the lower the better* rule. Our goal is to generate *lowest* ELCAs first. The key to this problem is the processing order of input nodes: if input nodes are sorted and processed in the document order, it is no doubt that generated ELCAs also follow the document order. Instead, we need to process nodes vertically, from lowest level to top level.

The basic idea of Join-based algorithm is as follows. Given nodes v_1, v_2 whose JDewey Sequences are S_1 and S_2 respectively, let l be the maximum number such that $S_1(l) = S_2(l) = N$, then node N at level l is the lowest common ancestor of v_1 and v_2. Consider two nodes $S_1 = 1.1.2.2.1$ and $S_2 = 1.1.2.3.2$ in Figure 1. Since $S_1(3) = S_2(3) = 2$ and $S_1(4) \neq S_2(4)$, node 2 at 3rd level is the LCA for S_1 and S_2. Starting from tails of S_1 and S_2, we scan JDewey Numbers from right to left. At some point, if two JDewey Numbers from two JDewey Sequences are the same, this number corresponds to the LCA for the two nodes.

3.1 "Pseudo" Algorithm

Consider two lists of nodes $L^1 = \{S^1_1, S^1_2, \ldots, S^1_m\}$, $L^2 = \{S^2_1, S^2_2, \ldots, S^2_n\}$ containing two keywords respectively. Let $l^1_m = max\{|S^1_1|, \ldots, |S^1_m|\}$ and $l^2_m = max\{|S^2_1|, \ldots, |S^2_n|\}$. Starting from $l = min\{l^1_m, l^2_m\}$, we retrieve all the JDewey Numbers at level l from two lists, i.e. $L^1(l) = \{S^1_1(l), \ldots, S^1_m(l)\}$ and $L^2(l) = \{S^2_1(l), \ldots, S^2_n(l)\}$. Then $L^1(l) \bowtie L^2(l)$ computes the ELCAs at level l. Recall that by property 2.1, $\forall l \in [1, min\{l^1_m, l^2_m\}]$, $L^1(l)$ and $L^2(l)$ are already sorted. Therefore, both merge join and index join are available for the join plan.

Algorithm 1 shows the pseudo code of the algorithm. The algorithm iteratively scans JDewey Numbers from two lists at each level and computes the common numbers. Those common JDewey Numbers correspond to the ELCAs at that level. Since the scan is bottom up, lowest ELCAs are generated first. Notice that those JDewey Sequences that are already descendants of generated ELCAs should be excluded from following processing (line 6,10,25 and 30) because of the ELCA semantics.

EXAMPLE 3.1. *Consider two-keyword query {XML, data}. Initial inverted lists are shown in Figure 3(a). Since $l^1_m = 7, l^2_m = 5$, 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 we move the next column, as shown in Figure 3(b), and join two lists of JDewey Numbers, i.e. $\{3, 5, 6\} \bowtie \{1, 2, 4\}$. Again no ELCA is generated. In Figure 3(c), join between $\{2, 3, 4, 5\}$ and $\{1, 2, 4\}$ finds matched numbers $\{2, 4\}$. So 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 reaches the root level, and finally identifies root as the last ELCA.*

Astute readers may already notice that in the above example, at root level, two results would be generated, if we follow the join semantics in the relational databases. This is because two nodes (1.2.3 and 1.3.5.6) can be the *occurrences* of the keyword {XML} for the root. Therefore, join semantics needs to be modified in our scenario. More formally, in the join $L^1(l) \bowtie L^2(l)$, if there is a matched number N appearing C_1 times in $L^1(l)$ and C_2 times in $L^2(l)$, then only one result (node N at level l) is outputted and all the corresponding JDewey Sequences are removed from L^1 and L^2.

Algorithm 1: Join-based algorithm to compute ELCA

```
Input  : $L^1 = \{S^1_1, \ldots, S^2_m\}$, $L^2 = \{S^2_1, \ldots, S^2_n\}$
Output: $R_l, l = min\{l^1_m, l^2_m\}, \ldots, 1$, where $R_l$ is a list
         of ELCAs at level $l$
1  $H_1 \leftarrow \emptyset$, $H_2 \leftarrow \emptyset$;
2  for $l \leftarrow min\{l^1_m, l^2_m\}$ to 1 do
3      if $|L^1(l)| \approx |L^2(l)|$ then            /* merge join */
4          $j_1 \leftarrow 1$, $j_2 \leftarrow 1$;
5          while $j_1 \leq |L^1(l)|$ && $j_2 \leq |L^2(l)|$ do
6              if $j_1 \in H_1$ then
7                  $j_1 \leftarrow j_1 + 1$;
8                  continue;
9              end
10             if $j_2 \in H_2$ then
11                 $j_2 \leftarrow j_2 + 1$;
12                 continue;
13             end
14             if $S^1_{j_1}(l) < S^2_{j_2}(l)$ then $j_1 \leftarrow j_1 + 1$;
15             else if $S^1_{j_1}(l) > S^2_{j_2}(l)$ then $j_2 \leftarrow j_2 + 1$;
16             else
17                 $H_1 \leftarrow H_1 \cup \{j_1\}$, $H_2 \leftarrow H_2 \cup \{j_2\}$;
18                 $R_l \leftarrow R_l \cup \{S_{j_1}(i)\}$;
19             end
20         end
21     end
22     else if $|L^1(l)| << |L^2(l)|$ then    /* index join */
23         $j_1 \leftarrow 1$;
24         while $j_1 \leq |L^1(l)|$ do
25             if $j_1 \in H_1$ then
26                 $j_1 \leftarrow j_1 + 1$;
27                 continue;
28             end
29             binary search $S^1_{j_1}(l)$ in $L^2(l)$;
30             if $\exists j_2$, $S^2_{j_2}(l) = S^1_{j_1}(l)$ && $j_2 \notin H_2$ then
31                 $H_1 \leftarrow H_1 \cup \{j_1\}$, $H_2 \leftarrow H_2 \cup \{j_2\}$;
32                 $R_l \leftarrow R_l \cup \{S_{j_1}(i)\}$;
33             end
34             $j_1 \leftarrow j_1 + 1$;
35         end
36     end
37 end
```

For the query with $k(k > 2)$ keywords, the algorithm is the same, except that the initial value of l becomes $\min\{l^1_m, \ldots, l^k_m\}$ and at each level one join becomes $k-1$ joins. In this paper, join ordering is only determined by the sizes of input lists, from the shortest to the longest, though more advanced techniques from relational databases can be applied.

Two highlights of the algorithm are worth to be mentioned. Firstly, the algorithm does not read the whole JDewey Sequences from the disk at once. Instead, it reads JDewey Numbers "column by column". Notice that scan starts from $l_0 = min\{l^1_m, l^2_m\}$, because it is obvious that there is no ELCA at the levels lower than l_0. This would save disk

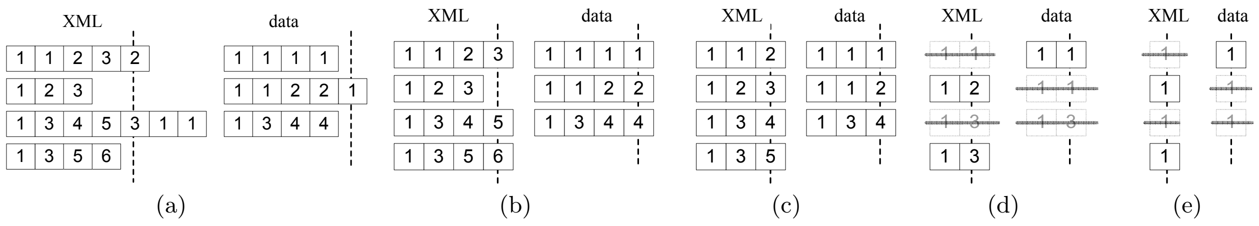


Figure 3: Execution example of top-K algorithm

I/O when the XML tree is deep and some keywords only appear at higher level. Secondly, unlike stack-based and index-based algorithms whose computation operations (and thus the complexity) are fixed regardless of keywords' frequencies, join-based algorithm chooses join plan dynamically (line 3 and 22) according to the relative sizes of lists. For example, after the first join, depending on the size of intermediate results, following joins may choose plan different from the first join. Moreover, the join plan is chosen based on column level, rather than the query level: join plan of next column may be totally different from current column.

Below we give the main-memory complexity of the algorithm. At each level, $k-1$ joins are performed, where k is the number of the keywords in the query. 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 list. There are altogether $min\{l_m^1,\ldots,l_m^k\}$ columns. For the worst case, all the keywords appear in the nodes at the lowest level. Then $min\{l_m^1,\ldots,l_m^k\}=d$ where d is the depth of the XML tree. Therefore, the whole complexity of the algorithm is: $O(d\cdot min\{\sum_{j=1}^{k}|L^j|, k|L^1|\log|L|\})$. For comparison, the complexity for stack-based algorithm and index-based algorithm are $O(d\sum_{i=1}^{k}|L^i|)$ and $O(dk|L^1|\log|L|)$. A major advantage of index-based algorithm is that when the frequency of one keyword is orders of magnitude lower than others, index-based algorithm is normally much faster than the stack-based algorithm because it avoids full scan of all input lists. However, for the case where all the keywords' frequencies are around the same, one scan of all the lists may be better. Our algorithm decides the join plan one the fly, and thus can achieve the optimality for queries with various frequencies.

The correctness of the join-based algorithm can be easily verified. Since the algorithm scans JDewey Numbers bottom up, ELCAs at lower level will be generated first. Also, since the algorithm filters out those sequences that are already descendants of generated ELCAs, ELCAs generated later will only take nodes as occurrences exclusively to themselves.

3.2 Index and Implementation

In the above subsection, we discuss a general join-based algorithm to compute ELCAs. However, implementation directly based on pseudo code in Algorithm 1 faces two challenges:

1. Same numbers may repeat many times in one column, especially at higher level, e.g. 3 and 1 repeats twice in $L^1(2)$ and $L^2(2)$. It not only increases the index size, but more importantly reduces the join efficiency, as in join operation scanning repeated numbers only generates at most one ELCA.
2. Since join are performed column by column, within each list, JDewey Sequences are expected to be stored by column as well. However, lengths of columns are variable because of variable lengths of JDewey Sequences. As in Figure 3(a), $L^1(7)$ contains only 1 number, whereas $L^1(1)$ contains 4 numbers. Variable-length columns make us lose row number information (a.k.a variable j_1, j_2 in Algorithm 1) which is used for filtering in the algorithm (line 6,10,25 and 30).

In this subsection, we discuss the implementation details and how to address these two problems efficiently.

3.2.1 Inferring Row Numbers

A naive solution to the above problems is: (1) fill all the empty positions in JDewey Sequences with padding values, which makes all the JDewey Sequences having the same length. (2) For repeated numbers (including both JDewey Numbers and padding values), it is straightforward to compress them using two variables N_i and C_i, where N_i is the value of the number and C_i is the count. So $[1,1,1,1]$ in a column is stored as $[1(4)]$.

In the join-based algorithm, we need the row information to filter out those JDewey Sequences that are already descendants of generated ELCAs. Therefore, given a specific JDewey Number in a compressed column, row numbers must be inferrable. The above solution works well for the merge join, because it scans all the columns from the head to the tail. Given the row number of the head, all the following row numbers can be easily inferred. However, it doesn't work for the index join. Index join locate the matched value by binary lookup. Without context information, row numbers cannot be inferred directly.

To overcome this problem, we replace the count C_i with row number of the next unique JDewey Number N_{i+1} in the column, and record the row number of the first JDewey Number in each disk block. When a JDewey Number is matched by binary lookup in a block, and we move t steps backward to see row number r, then the row number of the matched JDewey Number is $r+t-1$. For the worst case, we only need to scan the whole block.

EXAMPLE 3.2. *Assume the original column of JDewey Numbers is $V=[0,0,0,5,5,5,6,6,0,7,7,8,8,8]$. It is compressed as $V_c=[5,-7,6,-9,0,7,-12,8,-15]$ where negative numbers represent the row number. The row number of the head is $r_0=4$ because the first JDewey Number is in the 4th row. Assume matched value is 7. The nearest row number to its left is -9 which is $t=2$ steps backward. So the row number of 7 in original column is $9+2-1=10$. Furthermore, since the value following 7 is another row number, 7 also repeats in the original column. Recall that this value points to the*

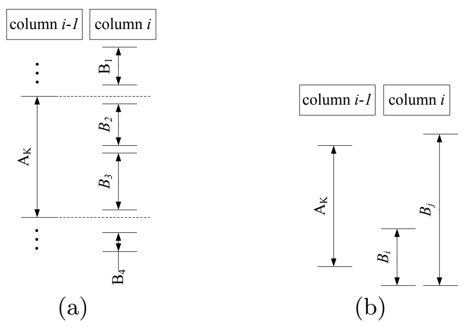


Figure 4: Range checking snapshot

row of the JDewey Number following 7 *(i.e. JDewey Number* 8*). So the largest row number of* 7 *is* $12-1=11$*. In other words, the range of JDewey Number* 7 *in this column is between row* 10 *and row* 11*.*

3.2.2 Range Checking

For the join of each column, Algorithm 1 (line 5,9,24 and 30) checks each row and filters out those JDewey Sequences that are descendants of existing ELCAs. In practice, since our column is compressed and same JDewey Numbers are grouped together, checking and filtering can be based on range rather than individual rows. Moreover, in implementation, checking is only performed when some JDewey Number is matched. The reason is that number of matched JDewey Numbers is normally much smaller than the input size. For performance purpose, this operation can be deferred until necessary.

Consider a snapshot shown in Figure 4(a). $B_j, j=1,\ldots,4$ are four ranges of JDewey Sequences in L^1 that are already occurrences of {XML} under generated ELCAs. A_k is the range of a JDewey Number N in column $l-1$ that can join with a value in the other list $L^2(l-1)$. By the ELCA semantics, range A_k should exclude all the occurrences of {XML} under existing ELCAs. Thus, those ranges within A_k in column l (i.e. B_2, B_3) should be excluded from A_k. In other words, if $|A_k|>|B_2|+|B_3|$, N is an ELCA that contains the occurrence(s) of {XML} after excluding occurrences under other ELCAs; otherwise, N is not an ELCA because it contains no occurrences of {XML} for itself, but only for other existing ELCAs.

Given a range A_k in column $l-1$, we only need to search those ranges within A_k in column l, and check the size of the ranges. Notice that the relationship between ranges in column l and A_k can only be either *contained* or *disjoint*. Cases in Figure 4(b) can never happen. This is simply because JDewey Numbers in B_i or B_j have the same parent. A_k either contain all of B_i and B_j or none of them. Having this property, range checking is only a binary search process (searching the ranges within A_k). When the join of column $l-1$ finishes, ranges within A_k (i.e. B_2, B_3) are replaced by A_k.

3.2.3 Index Structure

We design a two-level index structure: *master index (MI)* and *sequential index (SI)*. SI is purely a sequence of JDewey Numbers in compressed form, and MI is the index of SI. Figure 5 shows a fragment of the index structure, corresponding to L^1 in Figure 3(a).

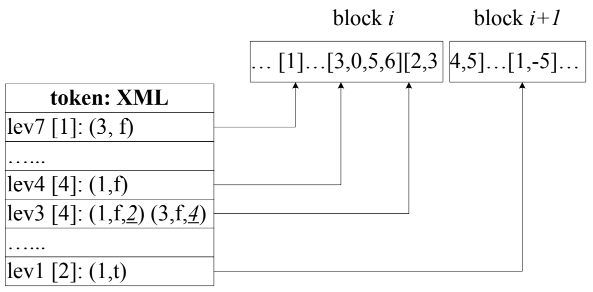


Figure 5: A fragment of two-level index

Each entry in MI corresponds a keyword/totken and contains metadata of each column. Each column in the entry points to the disk position where that column starts. Number within the square bracket is the number of values of a column after compression. Round brackets contain the metadata of the blocks the column spans. The first number in the round bracket is row number r_0 of the head of the column segment in this block. The second number is a boolean variable purely for the performance purpose: if the column segment in this block is uncompressed, and the matched value is at the position of p, then the row number of the value can be given directly by r_0+p without scanning backward. The third number is the first JDewey Number of the column segment in this block. (If the column spans only one column, then the third number is omitted.) For the index join, block heads are first looked up to locate the block where the value might be in. The located block is then read from disk for further binary search. If MI is cached in main memory, the I/O complexity of a single index join is $O(1)$ (if that disk block is not cached).

EXAMPLE 3.3. *In Figure 3(a),* $L^1(3)=\{2,3,4,5\}$ *and spans two disk blocks as shown in Figure 5. Two heads of the two column segments are* 2 *and* 4*, and their row numbers are* 1 *and* 3*. Also, this column is not compressed. So the metadata for the blocks are* $(1,f,2)$ *and* $(3,f,4)$*. If the searching value is* 5*, two block heads* $\{2,4\}$ *are first looked up, and second block is located. Then binary search within this block matches* 5 *at the position* 2*. Since column in this block is not compressed, its row number is given directly by* $3+2-1=4$ *without backward scanning.*

4. ALGORITHM FOR TOP-K RESULTS

In Section 3, we discussed the algorithm to compute complete results. The order of generated results follow the depth of the tree, from lowest to highest, which normally reflects people's expectation of more detailed results. Other than that, people may also want to incorporate other factors in ranking. For example, IR score evaluates the relevance between the content of the node and query; link analysis score evaluates the importance of a node, independent of the query. In such case, top-K algorithm becomes more important, because people normally only cares about a very few top ranked results.

4.1 Formal Definition of the Ranking Function and Constraints

Before discussing algorithm details, we first formally define the ranking score of ELCAs and the constraints the ranking functions are expected to satisfy. Given a keyword w, $g(v, w)$ is a global function that assigns v a ranking score evaluating its relevance with respect to w. $g(\cdot)$ can take any factors into account, e.g. IR score, link-based importance or the depth in the tree, and combine them in an arbitrary way.

Consider a query with k keywords $w_1, \ldots, w_k$. Assume node $\bar{v}$ is an ELCA at the level $\bar{l}$, and v^i at the level l^i is the occurrence of w_i under $\bar{v}$, $i = 1, \ldots, k$. $F(\cdot)$ is function that combines $g(v^i, w_i)$, $i = 1, \ldots, k$ into a score of $\bar{v}$.

Ranking Function $F(\cdot)$ combines $g(v^i, w_i), i = 1, \ldots, k$ with *damping factor*:

$$score(\bar{v}) = F\left(g(v^1, w_1) \times d(l^1 - \bar{l}), \ldots, g(v^k, w_k) \times d(l^k - \bar{l})\right)$$

$d(\cdot)$ is a decreasing function that reduces the importance of the occurrence of the keyword as its vertical distance to $\bar{v}$ increases. It reflects the intuition that compact results are more preferable because of the tighter relationship between keywords.

Notice that if v contains more than one occurrences of w_j, i.e. $v_1^j, \ldots, v_m^j$, $F(\cdot)$ only takes the maximum score (with damping factor) of the occurrences as input, i.e. $max\{g(v_1^j, w_j) \times d(l_1^j - \bar{l}), \ldots, g(v_m^j, w_j) \times d(l_m^j - \bar{l})\}$.

The combining function $F(\cdot)$ are expected to satisfy the following constraint:

Monotonicity v_1 and v_2 are two ELCAs at level $\bar{l}_1$ and $\bar{l}_2$ respectively. Their occurrences of the keywords are v_1^i and v_2^i $(i = 1, \ldots, k)$. If $\forall i \in [1, k]$, $g(v_1^i, w_i) \times d(l_1^i - \bar{l}_1) \leq g(v_2^i, w_i) \times d(l_2^i - \bar{l}_2)$, then $score(v_1) \leq score(v_2)$.

Monotonicity is the assumption the following top-K algorithm replies on. It is also true for most existing ranking functions. In the following discussion, we simply assume $F(\cdot)$ is the sum function, i.e. $score(\bar{v}) = \sum_{i=1}^{k} g(v^i, w_i) \times d(l^i - \bar{l})$.

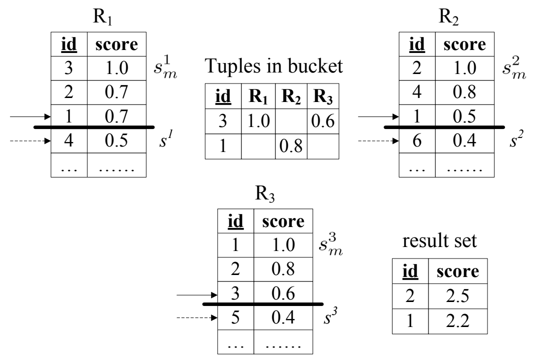
4.2 Review of Top-K Join Algorithm in Relational Databases

Top-K join problem in the relational database has been addressed by [17, 12]. The high level 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 between this tuple and all the tuples seen from other relations is performed. At any time, a threshold for all the unseen results can be estimated. Generated results whose scores are greater than the threshold can be outputted without blocking.

Consider the following SQL query where three relations are already sorted by scores, as shown in Figure 6.

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 scores. Figure 6 shows a snapshot of 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, join between $(4, 0.5)$ and



R1

id	score
3	1.0
2	0.7
1	0.7
4	0.5
…	……

Tuples in bucket

id	R1	R2	R3
3	1.0		0.6
1		0.8	

R2

id	score
2	1.0
4	0.8
1	0.5
6	0.4
…	……

R3

id	score
1	1.0
2	0.8
3	0.6
5	0.4
…	……

result set

id	score
2	2.5
1	2.2

Figure 6: Snapshot of relational top-K join

tuples seen from R_2 and R_3 is performed, and newly generated results are put into result set.

Let s^i denote the score of 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 scores of all unseen results are bounded by $max\{(s^1 + s_m^2 + s_m^3), (s_m^1 + s^2 + s_m^3), (s_m^1 + s_m^2 + s^3)\}$, i.e. $max\{2.5, 2.4, 2.4\} = 2.5$. Score of result tuple $(2, 2.5)$ is no less than the threshold, and thus can be outputted, whereas $(1, 2.2)$ is still blocked.

4.3 Top-K Algorithm for XML Keyword Search

In XML keyword search, generating ELCA is a join process and thus intuitively can directly apply the top-K join algorithms from relational databases. However, those top-K join algorithms are designed for general join patterns. In our scenario, the join pattern is only the *star* join, i.e. $R_1.a = R_2.b = R_3.c \ldots$, instead of the *sequence* join, i.e. $R_1.a = R_2.b_1$ AND $R_2.b_2 = R_3.c_1$ AND .... Given the property of the star join, there is an opportunity for further improvement on the threshold estimation. Furthermore, ranking scores of JDewey Sequences are decreased by damping factor when we move to higher level. There would be no unique order of JDewey Sequences in L^i. In the following, we address these two problems separately.

4.3.1 Top-K Algorithm for Star Join

Consider k-relation join $R_1.id = R_2.id = \ldots = R_k.id$. The new 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 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, add the score of t^i to t^0. If t^0 has been matched $k - 1$ times (there is no match when put into the bucket first time), move it from the bucket to the result set.

The threshold of unseen results for star join is estimated 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). In other words, the corresponding tuples are already in the bucket.

- For case 1, their upper bound can be estimated as $\sum_{i=1}^{k} s^i$.
- For case 2, tuples within bucket be grouped into $2^k - 2$ groups $G_P, P \subset \{1, \ldots, k\}$. All the tuples in G_P have

been seen in $R_j, j \in P$. Let $ms(P)$ denote the maximum score of tuples in G_P. Then the upper bound of tuples in G_P is $ms(P)+\sum_{j\notin P} s^j$. The upper bound of the whole bucket is: $max_{P\subset\{1,\ldots,k\}}(ms(P)+\sum_{j\notin P} s^j)$.

Since $ms(P) + \sum_{j\notin P} s^j \geq \sum_{i\in P} s^i + \sum_{j\notin P} s^j = \sum_i^m s^i$, we only need to consider the upper bound of tuples in the bucket. Therefore, the upper bound of unseen results is estimated by $max_P(ms(P) + \sum_{j\notin P} s^j)$, $P \subset \{1,\ldots,k\}$.

Note that for the top-K join algorithm for general join, upper bound estimation is: $max_i(s^i + \sum_{j\neq i} s_m^j)$ where $i = 1,\ldots,k$. $\forall P \subset \{1,\ldots,k\}$, $ms(P) + \sum_{j\notin P} s^j \leq \sum_{j\in P} s_m^j + \sum_{j\notin P} s^j \leq s^i + \sum_{j\neq i} s_m^j$ where $i \notin P$. Therefore, for the star join, the above algorithm provides tighter upper bound estimation for the unseen results. In Figure 6, if we use the new algorithm for star join, two tuples are in the bucket, $G_{\{1,3\}} = (3, 1.0+0.6) = (3, 1.6)$ and $G_{\{2\}} = (4, 0.8)$. If they can be results in future, their score would be bounded by $1.6 + s^2 = 1.6 + 0.4 = 2.0$ and $0.8 + s^1 + s^3 = 1.7$ respectively. Thus, the second result $(1, 2.2)$ can also be outputted without blocking. The reason of tighter upper bound is that we maintain those *partial results*, i.e. tuples that *partially joined* within a subset of relations. To estimate the upper bound of partial results, only scores from those unjoined relations are estimated.

In the algorithm, we maintain $2^k - 2$ groups within the bucket, which seems exponential to the query size (number of relations). However, notice that number of tuples maintained in the bucket is bounded by the number of tuples seen so far. Recall that each time new tuple is newly retrieved, all new valid join combinations with tuples seen from other relations are generated. Thus, we have to maintain the pool of all tuples already retrieved. In other words, grouping within bucket doesn't increase the algorithm complexity. In implementation, more groups only increase the overhead of upper bound estimation. For queries with a large number of keywords, this problem can be solved in two dimensions. In first dimension, we limit the number groups within the bucket: for k relations, we do not maintain all G_P's, $P \subset \{1,\ldots,k\}$, but only those P's such that $|P|$ is less than a threshold p_0. For those tuples in bucket that used to belong to P', $|P'| > p_0$, we randomly assign them to one group G_P where P is the largest maintained subset such that $P' \supset P$. Essentially, this modification balances a trade-off between cost and accuracy: it reduces upper bound estimation overhead, but increases the upper bound of unseen results. In second dimension, we break one bucket into two, corresponding to $R_1 \bowtie \ldots \bowtie R_{\lceil \frac{k}{2} \rceil}$ and $R_{\lceil \frac{k}{2} \rceil+1} \bowtie \ldots \bowtie R_k$ respectively. Tuples generated from two joins (buckets) are further joined. In such case, number of groups we maintained reduces to $2 \cdot (2^{\lceil \frac{k}{2} \rceil} - 2)$.

4.3.2 Top-K algorithm for XML Keyword Search

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 relation between $g(v_1, w) \times d(l_1 - l_2)$ and $g(v_2, w)$ is unknown before actual computation and comparison. In Figure 7, although the original score of the first JDewey Sequence is greater than the second, for the 4th column, the relation between $0.5 \times d(3)$ and 0.44 can be either greater than, equal to, or less than, depending on how fast the original score de-

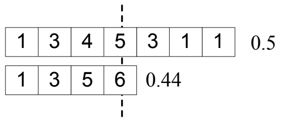


Figure 7: Two JDewey Sequences with ranking score

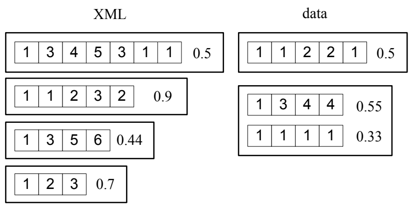


Figure 8: JDewey Sequences grouped by their lengths

creases. This fact means that for the JDewey Sequence list L^i, $L^i(l_1)$ and $L^i(l_2)$ can have different orders of JDewey Sequences with respect to their ranking scores.

To remedy this problem, JDewey Sequences in L^i are grouped by their lengths, as shown in Figure 8. For the JDewey Sequences' within one group, $\forall S_1, S_2$, if $score(S_1(l)) > score(S_2(l))$, $score(S_1(l - l_0)) > score(S_2(l - l_0))$. So there is an unique order for JDewey Sequences in one group. The number of groups is at most the height of the XML tree. Basically, this scheme breaks L^i into segments each of which is ordered by the original ranking scores. Complete order of the column can be easily given by merging segments online. In implementation, we maintain a cursor for each segment of L^i. Recall that the top-K algorithm only retrieves one JDewey Number from the column at one time. So at each iteration, the algorithms picks one JDewey Number with highest score from cursors and feed it into the bucket.

Similar to the algorithm generating complete ELCAs, the top-K algorithm joins JDewey Numbers column by column, from lowest to highest, to guarantee the ELCA semantics. The only difference is that for each column, top-K star join algorithm is used as join plan. After a join for one column, all the ELCAs at that level are generated. At any time, those generated results whose ranking scores are greater than the upper bound of unseen results can be outputted without blocking. However, notice that the algorithm in Section 4.3.1 is for one join and only gives the upper bound estimation of unseen results within that column. Since ELCAs can be generated by all the columns, we need to estimate upper bound of unseen results in other columns as well. More precisely, if we are currently performing join for column l_0, $\forall l < l_0$, we also need to estimate the upper bound of ELCAs at level l. Upper bound of ELCAs at level l can be estimated as: $\sum_{i=1}^k s_m^i(l)$ where $s_m^i(l)$ is the maximum score of column l in list $L^i(l)$ 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 $|S| = l$ and $score(S) = s_m^i(l)$, then we can skip column l directly. This is because: if the above two conditions are true, $\forall i \in [1, k]$, $s_m^i(l) = s_m^i(l+1) \times d(\cdot) < s_m^i(l+1)$. Therefore, upper bound of unseen results in column l is always less than unseen results in column $l+1$. On the other hand, if $\exists S \in L^i$ such that $|S| = l$ and $score(S) = s_m^i(l)$, there is no damping factor for $s_m^i(l)$ and thus upper bound of this

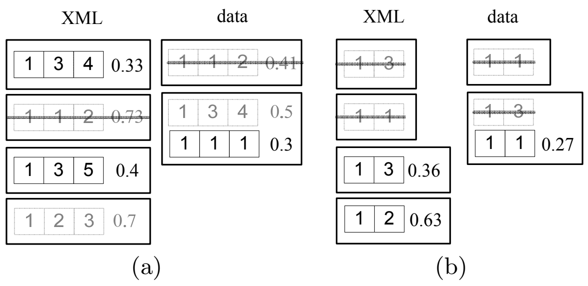


Figure 9: Snapshot of top-K algorithm

column must be computed.

EXAMPLE 4.1. *Consider again the query* {XML, data}. *Figure 8 shows the two lists* L^1, L^2 *and their original ranking scores. Assume the damping function is* $d(\Delta l) = 0.9^{\Delta l}$. *Joins for column 5 and 4 are first performed, and no result is generated. Figure 9(a) shows the status of the join for column 3. In the figure, two numbers from* $L^1(3)$ *(i.e.* $2, 3$*) and* $L^2(3)$ *(i.e.* $2, 4$*) have been retrieved and put into the hash bucket.* 2 *is matched as an ELCA and further moved into the result set. Its score is* $0.73 + 0.41 = 1.14$. *Similar to Section 4.3.1, upper bound of unseen results within column 3 is estimated as the maximum of two values: estimation of number* 3 *from* L^1 *(i.e.* $0.7 + s^2(3) = 0.7 + 0.3 = 1$*), and estimation of number* 4 *from* L^2 *(i.e.* $0.5 + s^1(3) = 0.9$*). Thus, upper bound of unseen results within column 3 is* 1*, which is less than node* 2*'s score. We also need to consider the unseen results in other columns, i.e. column 1 and column 2. Since both* L^1 *and* L^2 *do not contain sequence* $S, |S| = 1$*, we only need to consider column 2. The maximum scores from* $L^1(2)$ *and* $L^2(2)$ *are* $0.7 * 0.9 = 0.63$ *and* $0.5 * 0.9 = 0.45$. *So upper bound of unseen results in column 2 is* $0.63 + 0.45 = 1.08$, *which is also less than node* 2*'s score. Therefore, node* 2 *at level 3 can be outputted.*

When all the numbers in $L^1(3)$ *and* $L^2(3)$ *are retrieved, node* 4 *is also identified as an ELCA and its score is* $0.33 + 0.5 = 0.88$. *However, at this point, node* 4 *cannot be outputted. As in Figure 9(b), the upper bound results in column 2 is* $0.63 + 0.27 = 0.9$ *which is greater than node* 4*'s score.*

5. EXPERIMENTS

In this section, we experimentally evaluate proposed join-based algorithms on DBLP and XMark data set, and mainly compare our algorithms with two existing systems: stack-based [11] and index-based [29]. The size of XML document of DBLP 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 with a depth of 5, we group the papers firstly by conference/journal names, and then by years. Xcerse and Lucene are used to parse the XML tree and textual contents respectively. All the algorithms are implemented using Java under JDK 5. Experiments are performed on a Debian 2.40GHz PC with 1G memory.

5.1 Index Size

Table 1 shows the index sizes for different algorithms. As we can see, for algorithms generating complete ELCAs, the new node encoding which join-based algorithms reply on does not introduce much space overhead (or even saves spaces for DBLP data set), though it normally requires more bytes to encode a single node. This can be due to several reasons: firstly, in actual XML databases, some portion of nodes have a large number of siblings. It means that although traditional Dewey Id only requires uniqueness among siblings, number of bytes required for the corresponding level is still large. Secondly, in our index, compression is applied when lists of JDewey Sequences are stored by columns. To our observation, compression is highly effective for higher levels. For a node at higher level, even number of its siblings is very small, the number assigned to this node as the identifer among its siblings has to appear in all the Dewey Ids of its descendants. In contrast, in our index, this node's identifier only appears once in one inverted list of a keyword.

Index size for index-based algorithm is extremely large. This is because the implementation in [28, 29] uses a single B-tree in BerkeleyDB. It means that each key entry in the B-tree contains the keyword and Dewey Id. If the length of inverted list for a keyword is n, then this keyword repeats n times in the B-tree, which is a huge waste.

The lower half of Table 1 shows the index sizes for top-K algorithms. In top-K scenario, our algorithm has a great advantage in terms of space. RDIL, as mentioned in Section 1.2, requires both inverted lists and B+-tree because it retrieves new node from one list sorted by the score and then looks up other lists by B+-tree. This inevitably introduces much space overhead. On the other hand, the core operation of our top-K algorithm is hash join, and thus does not require any additional index.

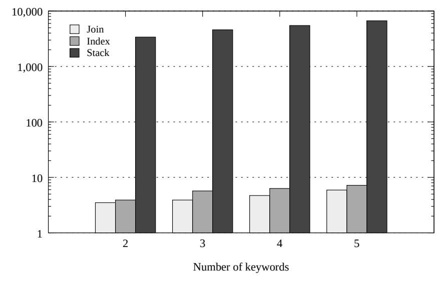
Table 1: Index Size for Different Algorithms

	DBLP		XMark	
Join-based	MI	SI	MI	SI
	14MB	327MB	4MB	302MB
stack-based	392MB		267MB	
index-based	2.1G		1.3G	
Top-K Join	MI	SI	MI	SI
	14MB	394MB	4MB	351MB
RDIL	inv. list	B+-tree	inv. list	B+-tree
	392MB	446MB	267MB	252MB

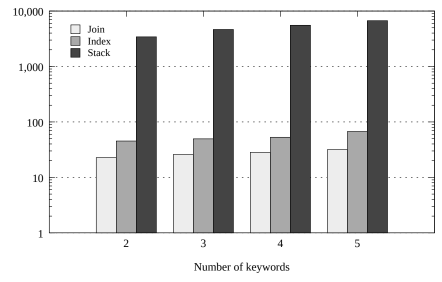
5.2 Query Performance for Complete ELCAs

We compare the query performance of three algorithms for complete ELCAs, varying both keyword frequencies and number of keywords. For each experiment, twenty queries within each frequency range are randomly selected. Execution time in the figures is the average of the twenty queries executed 5 times. Furthermore, experiments are on hot cache. For index-based algorithm in [29], BerkelyDB already provides a cache mechanism. For join-based algorithm, index is a sequential file, and our implementation replies on the file system cache directly. Note that in Table 1, Master Indices (MI) for both data sets are fairly small. Thus, MI is always cached in main memory. Stack-based algorithm, on the other hand, does not benefit a lot from cache, because it always needs to scan all the input lists.

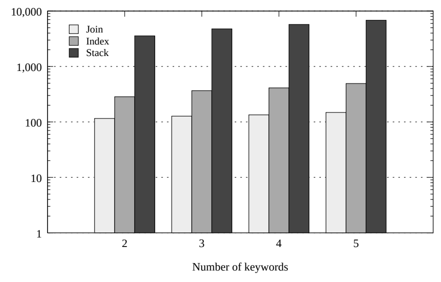
Experiment results are shown from Figure 10(a) to Figure 10(d). Due to the space limit, only results from DBLP are listed. Results from XMark are highly similar. In fact, query execution time mainly depends on two factors: keyword frequencies and keyword correlations. We vary number



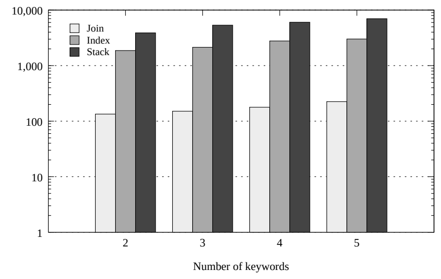
(a) Low Frequency: 10



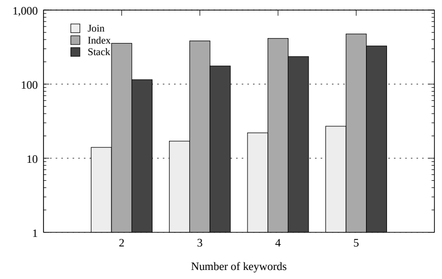
(b) Low Frequency: 100



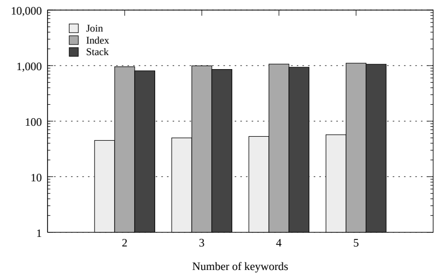
(c) Low Frequency: 1k



(d) Low Frequency: 10k



(e) Fixed Frequency: 1k



(f) Fixed Frequency: 10k

Figure 10: Query Performance For Complete Results

of keywords from 2 to 5. In all queries, highest frequency is fixed, i.e. 100k. Low frequency varies from 10 to 10k. As we can see from the figure, when low frequency is extremely low (10 or 100), execution time of our algorithm and 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 10(d). For the queries in that range, our algorithm already switch to the merge join. In fact, if we force the query plan to index join, the performance can be also as low as index-based algorithm. Furthermore, index-based algorithm implementation replies on a single B-tree. Though BerkeleyDB provides cache mechanism buffering top level internal nodes, there is a large amount of waste, as explained in last subsection, making index lookup less efficient. That is also why for queries with medium frequencies (Figure 10(c)), the performance difference between two algorithms is obvious, even join-based algorithm still chooses index join plan. Execution time of stack-based algorithm is always in the same order of magnitude, regardless low frequency. This is because the algorithm needs to scan all the input lists, and in the above experiments highest frequency is fixed.

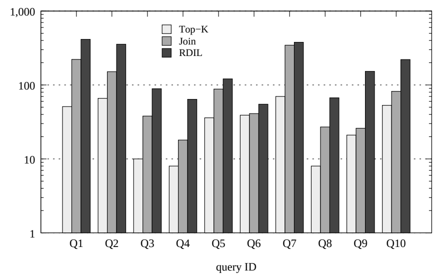
We also evaluate algorithms on keywords with same frequencies, as shown in Figure 10(e) and 10(f). Stack-based algorithm then performs slightly better than index-based algorithm, which can also be seen from their theoretical complexities. In these experiments, join algorithm performs much better stack-based algorithm, though their theoretical complexities seem same (for merge join). This can be contributed to several reasons: in Section 5.1, we mentioned that numbers of nodes at higher level must appear in each Dewey Ids of their descendants. When nodes are pushed into stack one by one, stack-based algorithm actually matches these common numbers and groups them online. In contrast, our index already compresses these common numbers into one value and thus saves online computation. Furthermore, even all input lists have the same size, the join plan may not stick to merge join. If the size of intermediate results is very small (for multiple joins), later joins will switch to index join. In fact, this happens often, because at lower levels, number of ELCAs is usually very small.

5.3 Query Performance for Top-K ELCAs

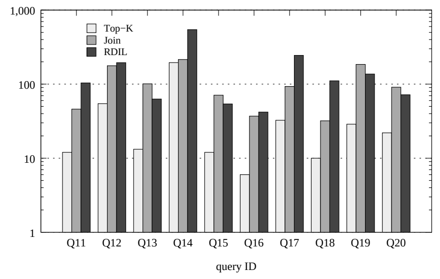
Now we evaluate top-K algorithm. We first run the algorithm on queries randomly generated in previous experiments. However, the performance of top-K algorithm is generally worse than the join-based algorithm generating complete results. The key is the keyword correlation issue. Conceptually, top-K join algorithm only performs well when the number of results is fairly large. For the keywords with low correlation, it normally takes very long scan to generate one result. In such case, there is no way for top-K algorithm to beat join-based algorithm optimized for complete results.

To overcome this limitation, we manually picked a set of queries with high keyword correlation, such as {sensor, network}, {XML, keyword, search}, and run three algorithms for comparison: top-K algorithm proposed in this paper, join-based algorithm generating complete results and RDIL. The results are shown in Figure 11. Overall, top-K algorithm is effective for keywords with high correlations: for most queries, the algorithm can terminate much earlier than join-based algorithm. RDIL is much less effective in terms of top-K processing. RDIL retrieves a new node from one list with highest score among all unseen nodes, and lookups other lists to compute ELCA. The problem is that the longest common prefix of Dewey Id of this node with nodes in other list can be very short. In the ranking function, damping factors will penalize it a lot, even its original score is very high.

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



(a) First Set of Correlated Queries



(b) Second Set of Correlated Queries

Figure 11: Query Performance For Top 10 Results

5.4 Discussion on Combined Index

Since algorithms for complete results and top-K results have strengths in different direction, it is straightforward to come with a combined index, building score index on top of L^i ordered by the JDewey Sequence. While this approach increases the index size, it makes three join plans (merge join, index join, top-K join) all available. Whether choose top-K join or not will be mainly based on join cardinality estimation: top-K algorithm should only be used for current column when the result size is estimated to be very large. Moreover, join plan can be chosen based on column level and other factions can also be taken into account. Recall that we need to estimate upper bound of results in all columns in top-K algorithm. If upper bound of some higher level is overwhelming large (imagine that in Figure 1 user's query exactly matches the title of the book), all the results below that level will be blocked. In that sense, there is no reason to perform top-K join for lower levels any more.

Note that in the join context, join cardinality estimation is an explicit problem and there are already many approaches in the literature. In [11], a hybrid algorithm is also proposed, combining stacked-based algorithm with RDIL. However, it faces the problem that it is unclear how to estimate cost and choose the right algorithm, which greatly limits its practical usage.

6. RELATED WORK

Keyword search in XML attracts much attention. First set of work, e.g. [11, 28, 29, 20, 25, 18], takes LCA's variations (e.g. ELCA, MLCA, SLCA) as query semantics and proposes different algorithms computing results. Major intuition of these algorithms is similar to stack-based [11] and index-based [29] algorithms, and thus their algorithm complexities are the same as these two. Join-based algorithm in this paper can also be adapted those similar semantics. Another set of work, e.g. [3, 2, 26, 18, 4], tries to extend the XQuery with keyword search operators, combining both IR ranking mechanism and tree structure information to improve search effectiveness.

Keyword proximity search in XML [14, 16] shares many similarities with LCA-based keyword search. However, there are some major differences distinguishing them. Firstly, LCA semantics may lead to conflict with proximity, as analyzed in the beginning of this paper. Secondly, there is no *exclusion* requirement for keyword proximity search: unlike most LCA semantics which require one node can only appear in one result, in keyword proximity search, different results can share some common structure. More generally, keyword proximity search takes the XML tree (with references) as a general graph. Hierarchy structure isn't considered much in these applications.

Mostly recently, much work has been done in new areas in XML keyword search scenario. For example, [24] studies query evaluation over virtual views of XML. The major difference with LCA keyword search is that given the view definition, the returned elements in [24] is fixed, while the returned results of LCA-based semantics can be arbitrary elements in XML tree. [20, 21] study the problem of how to return results with more semantics. They firstly computes SLCAs using index-based algorithm, and then further infers relevant results by analyzing match patterns and XML structures.

Beside semi-structured data, there is also much work on keyword search over structured data. DISCOVER[15], DBXplorer[1] and BANKS[7] are first three systems presented to support keyword search in relational databases. Their query semantics are similar: the query is a set of keywords and the results are sets of tuples that contain all the keywords and can be connected through the primary and foreign keys. Later work generally follows this semantics and further focuses on two aspects: efficiency and effectiveness. [12] incorporates IR-style ranking and proposes algorithms to return top K results efficiently. [19, 22] focus on the effectiveness and take into account more IR heuristics in the ranking function. [22] also proposes a Top-K algorithm which handles with non-monotonic ranking function.

Top-K query in relational databases has been widely studied for a couple of years. Existing work attacks the problem from different dimensions: monotonic ranking functions [10, 8, 6, 23], non-monotonic ranking functions [27, 22], existence of materialized views [13, 9, 5]. These work mainly focuses on the functions that combine multiple values from attributes of relation(s) and doesn't involve other operations. More related work to our scenario is the Top-K join problem[17, 12], which considers the traditional SQL join semantics. In this paper, we convert keyword search into relational join. Thus, there exists possibility to exploit more top-K processing techniques from relational database in XML keyword search scenario.

7. CONCLUSION

In this paper, we addressed the problem of keyword search in XML databases, under the ELCA semantics. By intuition, users tends to prefer those lower ELCAs because they reflect more specific information and data. However, none of existing algorithms provide efficient support for this intuition due to the fact that the order of generated results follows the document order. We proposed a join-based algorithm to compute complete ELCAs efficiently. We demonstrated the superiority of our algorithm over existing algorithms in both theoretical analysis and experimental results. Essentially, join-based algorithm is *orthogonal* to stack-based and index-based algorithms: in visual perception, join-based algorithm generates results vertically, from bottom level to top level, whereas stack-based and index-based algorithms generate results horizontally, from leftmost subtree to rightmost subtree; in implementation, join-based algorithm joins node encodings (JDewey Sequences) by column, whereas stack-based and index-based algorithms "join" (computing longest common prefix) node encodings (Dewey Id) by row. We also presented a top-K algorithm for one type of ranking functions. The algorithm adopts the idea from top-K join algorithms in relational databases and is optimized for XML keyword search scenario.

A more fundamental contribution of this paper is that we revealed a promising point where the relational join and XML keyword search intersect. Many techniques in the relational databases can be migrated to the XML keyword search scenario. For example, our system can decide the join plan on the fly and achieve optimality for queries with various frequencies.

8. REFERENCES

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

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

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

[4] S. Amer-Yahia and M. Lalmas. Xml search: languages, inex and scoring. *SIGMOD Record*, 35(4):16–23, 2006.

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

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

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

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

[9] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

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

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

[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

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

[14] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[15] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[16] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on xml graphs. In *ICDE*, pages 367–378, 2003.

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

[18] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

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

[20] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[21] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

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

[23] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72, 2006.

[24] 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*, pages 1057–1068, 2007.

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

[26] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *VLDB*, pages 625–636, 2005.

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

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

[29] Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.