XKMis: Effective and Efficient Keyword Search in XML Databases

Jiang Li Junhu Wang School of Information and Communication Technology
Griffith University, Gold Coast, Australia
Jiang.Li@student.griffith.edu.au, J.Wang@griffith.edu.au

Maolin Huang
Faculty of Engineering and Information Technology
The University of Technology, Sydney, Australia
maolin@it.uts.edu.au

ABSTRACT

We present XKMis, a system for keyword search in xml documents. Unlike previous work, our method is not based on the lowest common ancestor (LCA) or its variant, rather we divide the nodes into meaningful and self-containing information segments, called minimal information segments (MISs), and return MIS-subtrees which consist of MISs that are logically connected by the keywords. The MIS-subtrees are closer to what the user wants. The MIS-subtrees enable us to use the region code of xml trees to develop an algorithm for the search which is more efficient especially for large xml trees. We report our experiment results, which verify the better effectiveness and efficiency of our system.

Categories and Subject Descriptors
H.2 [**Database Management**]: Miscellaneous

General Terms
Algorithms

Keywords
XML, keyword search

1. INTRODUCTION

Keyword search has long been used to retrieve information from collections of text documents. Recently, keyword search in databases re-attracted attention of the research community because of the convenience it brings to users - there is no need for users to know the underlying database schema or complicated query language. In this paper we study keyword search in xml databases, which consist of xml documents. Unlike a plain text document, an xml document contains rich structural information. Such information should be utilized both in determining the result to be returned and in making the search more efficient.

Obviously, returning the entire document containing the keywords is inappropriate in xml keyword search systems, because usually the document is large and users are only interested in fragments of the document. Previously proposed xml keyword search systems (e.g., XRank [5], MLCA [9], SLCA [13], GDMCT [7], and XSeek [10]) used the concept of *lowest common ancestor* (LCA), or its variant, to connect the xml nodes which contain the keywords and as the result to be returned. The basic idea is to find those nodes that contain some of the keywords, and return the subtree rooted at the lowest common ancestor of these nodes.

The LCA-based approaches have a common inherent problem, which is that they may link irrelevant xml nodes together and return large amounts of useless information to the user. This problem is called the *false positive* problem in [8]. Consider the data tree shown in Figure 1. Suppose the user wants to find papers in volume 12 which contain the word "Query", and he submits the query {volume, 12, Query}. The early LCA-based approaches (e.g., XRank, SLCA) will return the subtree rooted at SigmodRecord (0, 102, 0), i.e., the entire document to the user, because the node (0, 102, 0) is the lowest common ancestor of nodes (53, 55, 2), (54, 54, 3) and (33, 33, 5) which contain these keywords. In order to solve the false positive problem, Li et al [8] proposed the concept of *valuable* LCA (VLCA). Suppose there are two xml nodes u and v that match the keywords, and their LCA is w. If there exist nodes which have the same label on the paths $w \rightarrow u$ and $w \rightarrow v$, u and v are considered to be *Heterogenous* and will not be linked together. Consider the query {volume, 12, Query} again. On the paths SigmodRecord (0, 102, 0) → SQL Query (33, 33, 5) and SigmodRecord (0, 102, 0) → 12 (54, 54, 3), nodes issue (1, 51, 1) and issue (52, 101, 1) have the same label. Therefore, nodes SQL Query (33, 33, 5), volume (53, 55, 2) and 12 (54, 54, 3) are regarded as *Heterogenous* and not connected together. This approach solves the problem in some cases, but there are still many cases for which it won't help. Consider the query {Kurt, Software} issued on the *dblp* data tree in Figure 2 (a). Nodes $Kurt$ and $Software$ are linked together through the LCA *dblp* because there are no nodes with the same label on the paths *dblp*→ $Kurt$ and *dblp*→ $Software$. Hristidis et al. in [7] introduced the concept of *minimum connecting trees* (MCTs) to exclude the subtrees (rooted at LCAs) that do not contain keywords. This approach makes the results more compact, but it can not prevent unrelated nodes from being linked together. Re-

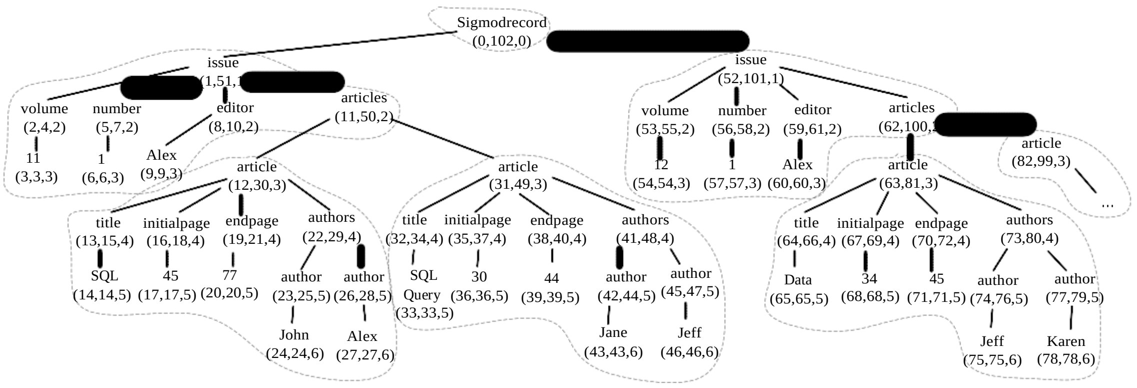


Figure 1: SigmodRecord data tree and minimal information segments

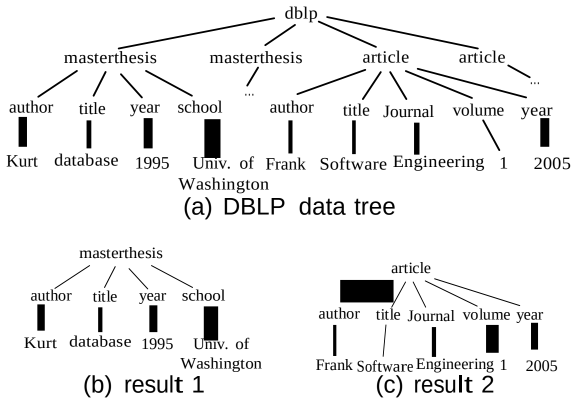


Figure 2: DBLP data tree and partial search results

cently, Liu et al [11] investigated the axiomatic approach to improve the relevancy of match nodes. They proposed two properties (i.e., *monotonicity* and *consistency*) that an xml keyword search engine should satisfy as well as an algorithm called MaxMatch with these properties. MaxMatch improves the quality of results, but it can not solve the false positive problem.

Another problem of existing LCA-based approaches is that the returned answers may contain too little information so they are not informative enough to users. For example, if a query {Johh} is issued over the data tree in Figure 1, the node *John* (24, 24, 6) will be returned. For the query {John, Alex}, the subtree rooted at node *authors* (22, 29, 4) will be returned because it is the LCA of the matched nodes. Obviously, such results do not provide users with much useful information. To make the answers more meaningful, XSeek [10] tried to recognize the possible entities and attributes in the data tree, distinguish between search predicates and return specifications in the keywords, and return nodes based on the analysis of both xml data structures and keyword match patterns. Xu et al. [12] proposed to use *minimal information unit (MIU)* in place of nodes and return the lowest common ancestor MIUs. An xml document is partitioned into a series of MIUs and they are treated as the minimal unit instead of xml nodes during processing. However, these proposals are LCA-based, so they suffer from the false positive problem mentioned above. For example, for the query {volume, 12, Query} discussed above, MaxMatch will return the tree shown in Figure 3 (a). The system described in [12] will return the subtree rooted at Sigmodrecord (0, 102, 0).

Our system is not LCA-based. In our system, we partition an xml document into a series of meaningful and self-containing segments, called *minimal information segments (MISs)*. These MISs are similar to the MIUs in [12], but unlike [12], we do not require the existence of a schema file, and we do not return the lowest common ancestor MISs, instead, we return MIS subtrees which consist of MISs logically connected by the keywords (see Section 3 for the definition of MIS subtree). For example, for the query {volume, 12, Query} against the datatree in Figure 1, our system will return two *partial match* results shown in Figure 3 (b) and (c). For the query {John, Alex}, our system will return the result shown in Figure 3 (d) rather than the subtree rooted at authors (22,29,4), and for the query {Kurt, Software} against the data tree in Figure 2, our system will return two partial match results as shown in Figure 2 (b) and (c). Overall, our system will significantly reduce the false positives and at the same time, make the returned result more meaningful. Furthermore, since we do not need to compute the LCA of nodes, we can use the region code [4] (rather than the Dewey code) of data trees in our search algorithm. This enables us to significantly reduce the number of stack operations required, making our search more efficient than the LCA-based approaches, especially for large xml documents.

The rest of the paper is organized as follows. Section 2 presents the xml data model and some basic notations. Section 3 presents the intuition and formal definitions of minimal information segment, MIS subtree, as well as the result of a keyword search. Section 4 then provides our new algorithm for finding the query results. Section 5 describes and analyzes our experiment results. We conclude the paper in Section 6.

2. PRELIMINARIES

An xml document is modeled as an unordered tree, called the *data tree*. Each internal node (i.e., non-leaf node) has a label, and each leaf node has a value. The internal nodes represent elements or attributes, while the leaf nodes represent the values of elements or attributes. Each node *v*

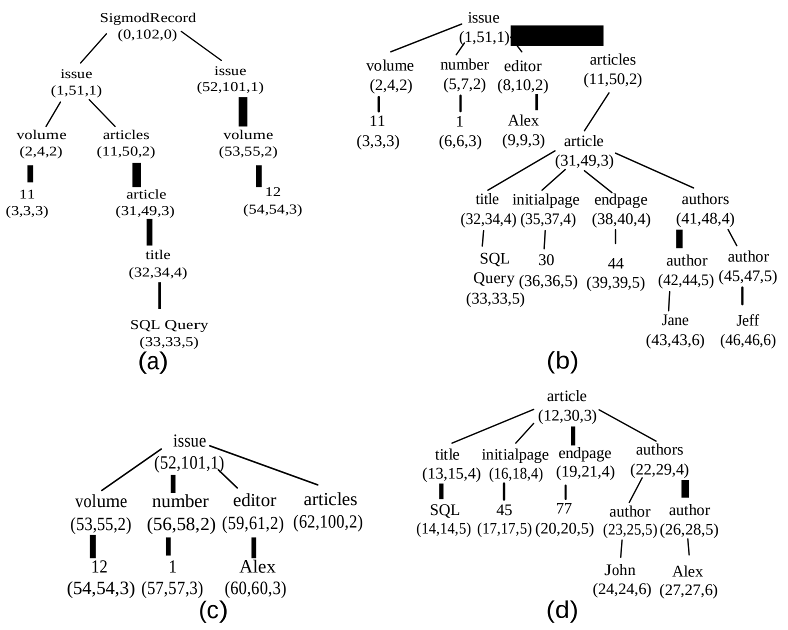


Figure 3: Example search results over the SigmodRecord data tree

in the data tree has a unique region code (*start, end, level*) [4], which acts as an identifier of that node (in fact, *start* and *end* uniquely identify the node). Such a coding scheme has several useful properties: (1) ancestor-descendant and parent-child relationships can be identified in constant time: for any two nodes v_1, v_2 in t, v_1 is an ancestor of v_2 iff $v_1.start < v_2.start \leq v_2.end < v_1.end$, and v_1 is the parent of v_2 iff it is the ancestor of v_2, and $v_2.level - v_1.level = 1$. (2) v_1, v_2 do not have ancestor-descendant relationship, and v_1 lies in a path to the left of the path where v_2 lies iff $v_1.end < v_2.start$. An example data tree is shown in Figure 1.

Let u and v be two nodes in a data tree. We will use $u \prec v$ to denote the fact that u is an ancestor of v. We will also use *root*(t) to denote the root of t.

A *keyword query* is a finite set of keywords $K = \{k_1, \ldots, k_n\}$. Given a keyword k and a data tree t, the search of k in t will check both the labels of internal nodes and values of leaf nodes.

3. MINIMAL INFORMATION SEGMENT AND ANSWERS TO KEYWORD QUERY

In this section, we first discuss the intuition. Then, we formally define the concept of minimal information segment (MIS). After that, we define MIS subtrees and answers to a keyword query.

3.1 The intuition

A keyword search system should return results that are both *informative* and *compact*. In other words, each result should not contain too little or too much information. To achieve informativeness, we divide the data tree into *minimal information segments (MISs)* which represent data about real-world objects. If a keyword occurs in a MIS, then the whole MIS, rather than the node containing the keyword, will be returned as part of the answer. This ensures that every node in the result is in a meaningful context. For example, the MISs in the data tree shown in Figure 1 are encircled by dotted lines. They represent the objects *sigmodrecord*, *issue*, *article* and so on. If the keyword John is submitted, then the entire MIS in which *John* appears, i.e., the MIS rooted at article (12, 30,3), rather than the node John (24,24,6) alone, will be returned to the user[1]. To achieve compactness, we do not link two keyword-containing MISs via their lowest common ancestor as in previous work. Instead, we only link them together if they have an ancestor-descendant relationship, or if they are both linked to a third keyword-containing MIS via ancestor-descendant relationships. This will make sure that only closely related MISs are linked together, hence each result contains information about closely related objects. Intuitively, if the xml document is well-designed, then MISs that have ancestor-descendant relationships are directly related, while those that do not have ancestor-descendant relationship are only loosely related. For example, in the SigmodRecord data tree, article objects that are descendants of an issue object represent articles published in that issue, while articles that are not descendants of an issue are not published in that issue. Therefore, we should only link articles nodes to an issue node which is the ancestor of the articles.

3.2 Formal definitions

In this work, we use *minimal information segments (*MIS*s)* to represent the objects in an xml document. A formal definition of MIS is given below.

[1]In practice, sometimes the MIS may be very large and contain unwanted information. To deal with such cases, we may select only the most important information contained in the MIS. This is left as part of our future work.

Definition 1. Let t be a tree and u be a node in t. The *full subtree* of t rooted at u, denoted t_u, is the tree consisting of u and all descendants of u. A *subtree of t rooted at u* is a tree obtained from t_u by removing zero or more full subtrees rooted at some descendants of u. A *lower subtree* of t is a subtree of t rooted at a descendant of *root*(t).

Note that the above definition of subtree catches any tree consisting of part or all of the nodes in t. But we deliberately stress that a subtree can be obtained by removing full subtrees from other full subtrees.

Definition 2. Let t be an xml tree. A node u in t is said to be a *simple node* if it is a leaf node, or has a single child which is a leaf node. A *minimal information segment (MIS)* in t is a subtree S of t with the following properties: (1) *root*(S) is either *root*(t), or has siblings with the same label as itself, and (2) *root*(S) is not a simple node, (3) no lower subtree of S is a MIS.

The MISs in the data tree shown in Figure 1 are encircled by dotted lines.

With Definition 2, some recognized MISs may still not be informative enough. For example, if the author element in Figure 1 contains first name and last name sub-elements, it will also be considered as a MIS. However, this MIS does not contain much information. In practice, we can set a *threshold* (e.g., the minimum number of elements in a MIS) to guarantee a MIS is informative enough.

Note that any node in t belongs to one and only one MIS, that is, no two MISs overlap in their nodes. Furthermore, for any two MISs S_1 and S_2, either there are no edges from nodes in S_1 to nodes in S_2 (when *root*(S_1) is not an ancestor of *root*(S_2), or when there exists MIS S_3 such that *root*(S_1) $<$ *root*(S_3) $<$ *root*(S_2)), or there is a single edge from S_1 to S_2 (when *root*(S_1) is an ancestor of *root*(S_2) and there is no MIS S_3 such that *root*(S_1) $<$ *root*(S_3) $<$ *root*(S_2)), or there is a single edge from a node in S_2 to a node S_1 . Therefore, if we treat each MIS as a node, and treat an edge from a node in S_1 to a node in S_2 as an edge from S_1 to S_2, then all such nodes and the edges between them form a tree, which we call the *MIS tree* of t, denoted MIS(t).

Definition 3. An *answer* (or *result*) of a keyword query K over a data tree t is a subtree S of MIS(t) with the following properties: (1) every node in S contains at least one keyword, (2) no lower subtree of S contains all the keywords in K, (3) every descendant m of *root*(S) in MIS(t) that contains *strictly part* of the keywords is in S, provided m is not in another answer S' such that *root*(S) $<$ *root*(S') and S' contains all of the keywords in K.

An answer S is said to be *optimal* if *root*(S) contains all of the keywords in K. It is said to be *sub-optimal* if the nodes in S collectively contain all of the keywords but *root*(S) alone doesn't. An answer is said to be a *partial match* if it is neither optimal nor sub-optimal.

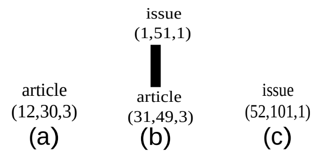


Figure 4: Search results over the SigmodRecord data tree, each node in a result represents a MIS

Example 1. *Consider the query {Alex, SQL} over the data tree in Figure 1. An optimal result, a sub-optimal result, and a partial match are shown in Figure 4 (a), (b) and (c) respectively. Note that the nodes in these answers are MIS nodes, each of which represents a collection of nodes in the original data tree.*

In the above example, it is most likely that the user wants information about articles written by Alex which contain "SQL" in the title. The optimal answer in Figure 4 (a) has only one MIS which contains all the keywords, which is indeed an article object the user probably expects. A relatively lower possibility is that the user wants information about articles whose title contains "SQL" and which is published in an issue edited by Alex. The sub-optimal answer in Figure 4 (b) reflects this requirement of the user. The partial match in Figure 4 (c) reflects the (less likely) possibility that the user is interested in an issue edited by Alex.

Discussion:

- Optimal results have the greatest chance to meet users' expectations. This is because, if an individual MIS contains all the keywords, then the nodes containing those keywords have the closest relationships.
- Sub-optimal results are a secondary choice compared with optimal results. Sometimes, two or more directly related MISs are needed to find all required data. This is similar to selecting data from multiple tables through joins in relational databases. In the query {volume, 11, SQL}, the issue entities of volume 11 and the article entities of SQL should be joined together and the join condition is the ancestor-descendant relationship between them. (Note: The join condition of ancestor-descendant relationship is important because it can prevent the articles of SQL from being linked with the issues which are not volume 11.)
- Partial match results are the last choice because each of them contains strictly part of the keywords. However, a partial match result also has its value especially when there are no or few optimal and sub-optimal results. It can provide users with useful clues and help users to refine their queries. Popular web search engines also return related partial match results but with lower ranks.

3.3 Partitioning a data tree into MISs

A straightforward method to partition the data tree t into MISs is as follows. First, we do a width-first traversal of the data tree to identify all those nodes in t which have the same label as some of its siblings and which are not simple nodes. The full subtrees rooted at these nodes are *potential* MISs. Then, in each of these full subtrees, we cut off those subtrees that are MISs themselves. Note that we do not need a schema file (such as DTD). If the schema file exists, the MISs can be more accurately identified and the partitioning process can be simplified, e.g., using the method in [12].

Region code for MIS tree and the modified coding scheme of data tree To facilitate the computation of MISs containing a given keyword, we modify the region code of the data tree so that all nodes within a MIS share the same region code. We can do this by simply replacing the region code of v, for each node v in t, with the region code of the root of the MIS in which v lies. For example, all nodes in the MIS rooted at issue (1, 51, 1) in Figure 1 will share the region code (1, 51, 1). We call the new region code of v the *modified region code*, and the original code of v the *normal region code*. The data tree in Figure 5 (a) uses the modified region code.

3.4 Further ranking of answers

As discussed above, generally an optimal answer is preferable to a sub-optimal answer, and a sub-optimal answer is preferable to a partial match. However, not all answers within the same class (optimal, sub-optimal, or partial match) are equally interesting to the user. Therefore, the answers to a keyword query need to be further ranked according to their degree of relevance. To this end, we classify all MISs in data tree t into different *MIS-types* according to their root label. Let S_1 and S_2 be two MISs in t. If the roots of S_1 and S_2 have the same label, we say S_1 and S_2 are of the same MIS-type.

Classifying MISs into different MIS-types is very similar to organizing data into different tables in relational databases. A MIS of a MIS-type is like a *record* in a table. Consider the data tree in Figure 1, each MIS of the *article*-type is like an *article record* in the *article* table. Therefore, it is possible to apply the ranking scheme in relational databases. The only difference is that they rank a joining tree of tuples, but we rank a MIS subtree of MISs. In this work, we chose to use a ranking function derived from [6]. Suppose T is a MIS subtree that contains keywords. The score $Score(T, K)$ of T is calculated using the formula below.

$$Score(m_i, K) = \sum_{k \in K \cap m_i} \frac{1 + \ln(1 + \ln(tf))}{(1 - s) + s\frac{dl}{avdl}} \cdot \ln(\frac{N + 1}{df}) \quad (1)$$

$$Score(T, K) = \frac{\sum_{m_i \in T} Score(m_i, K)}{size(T)} \quad (2)$$

where K is the keyword query; m_i is a MIS in T, tf is the frequency of keyword k in MIS m_i; N is the total number of MISs of m_i's MIS-type; df denotes the total number of MISs that contain keyword k; dl is the length of the text attribute of m_i; $avdl$ is the average length of the text attribute of m_i's MIS-type; s is a constant value (usually 0.2); and $size(T)$ is the size of a MIS subtree T. Different from the size of a joining tree, the size of a MIS subtree includes not only the number of MISs, but also the *level* of each MIS, as defined below:

$$Size(T) = \sum_{m_i \in T} 1 + (m_i.level - root(T).level) \quad (3)$$

For example, the size of the MIS subtree in Figure 4 (b) is $1 + (1 - 1) + 1 + (2 - 1) = 3$.

Algorithm 1 $XSMis(K)$

```
1: let $M_i$ be the stream of sorted MISs which contain the keyword $k_i$, for all $1 \le i \le n$
2: let $M = \{M_1, \ldots, M_n\}$
3: XKMis_Construct($K, M, n$)
4: XKMis_Output($OList, SList, PList$)
```

4. ALGORITHM

In this section, we present our algorithm for finding all answers to a keyword query K over data tree t. The main algorithm is shown in Algorithm 1. There are two steps involved. In the first step, it calls the procedure XKMis_Construct to construct the result lists *OList*, *SList* and *PList*, which contain the optimal, suboptimal, and partial match results respectively. In the second step, it calls XKMis_Output to output these results. Before explaining the algorithm in detail, we need to define some notions.

4.1 Notations

Given a keyword query $K = \{k_1, ..., k_n\}$, for each keyword k_i, there is a stream, M_i, consisting of all the MISs which contain K_i. The MISs in each stream M_i are arranged in ascending order of their *start* values. In our implementation, we have chosen to allow multiple occurrence of the same MIS in a stream: if keyword k_i occurs N times in MIS *mis*, then *mis* will appear N times in M_i (an alternative is to have an attribute in each MIS to record the number of times each keyword occurs in it). In addition, for each stream M_i, there exists a pointer P_i pointing to the current MIS in M_i. The function $Advance(M_i)$ moves the pointer P_i to the next MIS in M_i. The function $isEnd(M_i)$ judges whether P_i points to the position after the last MIS in M_i. The function $getMis(M_i)$ retrieves the current MIS of M_i.

To compute streams $M_1, \ldots, M_n$, we build a B^+-tree index on all of the words in t, each word k_i in the leaf of the B^+-tree points to the list of modified region codes of nodes containing k_i. Using this index and the modified region codes we can find the streams efficiently.

As mentioned above, the lists *OList*, *SList* and *PList* are used to store optimal results, sub-optimal results and partial match results respectively. Each item in these lists is a pointer which points to a list of MISs (Figure 5 (b)), which represents a MIS-subtree to be returned to the user. During processing, for each MIS m, in addition to the region code (*start, end, level*), we use four additional attributes $flag$, *optimal*, *score*, and *num*. The $flag$ attribute is a n-bit binary number, which indicates which keywords are contained in m. The function $SetFlag(flag, i)$ sets the i^{th} bit of the flag to 1, which means the keyword k_i is contained in m.

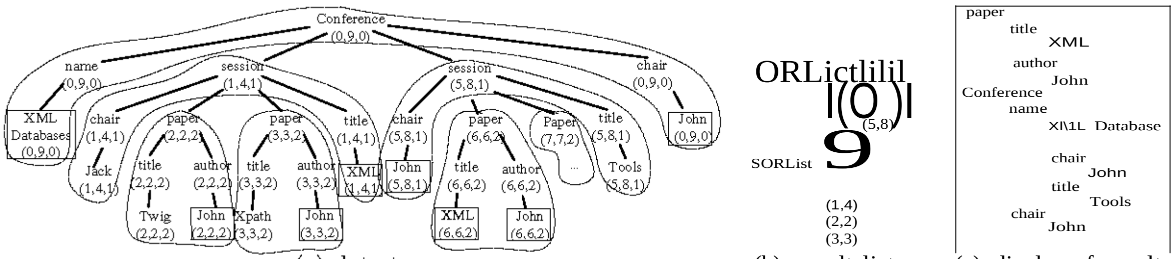


Figure 5: Example to explain algorithms

The function *countOnes(flag)* returns the number of l's in the flag. The *optimal* attribute is used to indicate whether a MIS subtree is an optimal result (contains all keywords). The *num* attribute records the total number of keywords contained in a MIS subtree (duplicate keywords are counted multiple times). The *score* attribute records the ranking score of a MIS subtree.

4.2 XKMis_Construct

Our algorithm for building the result lists, XKMis_Construct, is shown in Algorithm 2. After a MIS *m* is initialized, the procedure *getMinMatch* is repeatedly called (line 3, 4) to get the MIS, *mmin*, which has the smallest *start* value among all the MISs in *M* that have not been processed. For each MIS *mmin* retrieved, the indexes of the keywords contained in *mmin* are recorded in the *flag* attribute, and the number of keywords contained in *mmin* are recorded in the *num* attribute (line 22, 23). The loop will stop when all the MISs in *M* have been processed and there are no MISs left in the stack *S* (line 3). From line 5 to 7, we check whether the current MIS *m* is a descendant of *top(S)*, if not, we pop up *top(S)*, and pass it to procedure *M ovetoResultList* to see whether it should be appended to result lists and which result list it should be appended to. Procedure *M ovetoResultList* (line 25-39) does not directly append a MIS *mis* to a result list. It first checks whether *mis* contains all the keywords (line 26), if yes, the MIS will be appended to O*List* or *S List* depending on the value of *mis.optimal*. If *mis* does not contain all the keywords, and the stack *S* is empty, then *mis* will be appended to *PList*. If *mis* does not contain all the keywords, and the stackS is not empty, then *mis* will be linked with the MIS at the top of *S*, *mistop*, and *mis.flag* will be copied over to *mistop* (line 37). It should be noted that while copying the flag, the set of keywords contained in *mistop* is checked to determine whether its *optimal* attribute should be set to FALSE (line 41, 42). When the current MIS *m* does not have descendants that have not been checked (line 8), *m* is directly passed to procedure *M ovetoResultList* (line 9). If *m* has descendants that have not been checked (line 10), it is pushed into the stackS. Note that at any moment of time, the MISs in the stack are arranged in descendant-to-ancestor order (from top-down).

EXAMPLE 2. *Consider the keyword query {XML, John} and the data tree in Figure 5 (a). The first call of procedure getMinMatch returns the MIS (0, 9). The fiag attribute indicates it contains both ^{11}XML" and ^{11}]ohn", and in total contains 2 keywords (the value of num attribute is 2). Because the initial start value of the current MIS m is -1, m is not pushed into the stack S. At the end of the first loop, m is assigned with mmin (i.e. MIS (0, 9)). In the second loop MIS (1, 4) is returned by getMinMatch. The fiag attribute indicates it only contains ^{11}XML" and in total contains 1 keyword. Because this MIS is a descendant of the current MIS m =* (0, 9), *m is pushed into S and the current MIS m becomes (1, 4). Similarly, the next call of getMinMatch returns MIS (2, 2), which causes MIS (1, 4) to be pushed into S, and m becomes (2, 2). The next call of MIS getMinMatch returns (3, 3), which is not a descendant of the current MIS m =* (2, 2), *so m is directly passed to procedure MovetoResultList. MIS (2, 2) does not contain all the keywords, so it is linked with the MIS at the top of stack S (i.e. MIS (1, 4)). Similarly, MIS (5, 8} also forces MIS (3, 3} to be linked with MIS (1, 4). In the next loop, MIS (6, 6) is returned by getMinMatch. The current MIS m*(5, 8) *first makes top(S) (i.e. MIS (1, 4)) to be popped up. Because MIS (1, 4) contains all the keywords and the optimal attribute is false, it is appended to SList. Next, since MIS (6, 6) is a descendant of m =* (5, 8), *m is pushed into S. There are no MIS left in M, but the stack S is not empty. A MIS* (00,00) *is returned by getMinMatch. This infinite MIS makes the current MIS m =* (6, 6) *appended to OList because it contains all the keywords and the attribute optimal is true. Now m becomes* (00, 00), *which first causes MIS (5, 8} to be popped up and linked with top(S) (i.e. MIS (0, 9)) because MIS (5, 8) does not contain all the keywords. Then, MIS (0, 9) is also popped up and appended to OList because it contains all the keywords and the optimal attribute is true. The final result lists are shown in Figure 5 (b).*

Time complexity: XKMis_Construct scans each *Mi* only once to generate all the MIS-subtrees. In the worst case, each MIS retrieved from *Mi* needs to be pushed into and popped-up from stackS, which can be finished in constant time. Each MIS popped up from *S* is checked to see whether it contains all the keywords. The MIS that does not contain all the keywords is linked with the MIS on the top of the stack *S*. This can also be done in constant time. Therefore, the worst-case time complexity is O(IMI) (IMI is the total number of MISs containing keywords), which is linear in IMI

4.3 XKMis_Output

After the result lists are constructed with XKMis_Construct, the final results need to be shown to users. Algorithm 3 shows how to build the final results and display them to the user. In our algorithm, optimal results are displayed first,

Algorithm 2 XSMis_Construct(K, M)

Input: keyword query $K = \{k_1, ..., k_n\}$, a set M of streams
Output: result lists $OList$, $SList$ and $PList$

```
 1: Initialize a MIS $m$, set $m.start = -1, m.end = \infty$
 2: Create an empty stack $S$
 3: while $S = \emptyset$ OR $\neg isEnd(M_1) \wedge ... \wedge \neg isEnd(M_n)$ do
 4:     $m_{min} = getMinMatch(M, n)$
 5:     while $S = \emptyset$ AND $top(S).end < m.start$ do
 6:         $mis_{top} = pop(S)$
 7:         $MovetoResultList(mis_{top}, n)$
 8:     if $m_{min}.start > m.end$ then
 9:         $MovetoResultList(m, n)$
10:     else
11:         if $m.start = -1$ then
12:             $push(S, m)$
13:     $m = m_{min}$
14: procedure getMinMatch($M, n$)
15:     Initialize a MIS $mis$ set $mis.start = mis.end = \infty$
16:     for $i = 1$ to $n$ do
17:         if $getMis(M_i).start < mis.start$ then
18:             $mis = getMis(M_i)$
19:     Set $mis.flag = 0$, $mis.optimal = true$, $mis.num = 0$
20:     for $i = 1$ to $n$ do
21:         while $getMis(M_i).start = mis.start$ do
22:             $SetFlag(mis.flag, i)$
23:             $mis.num ++$
24:             $Advance(M_i)$
25:     return $mis$
26: procedure MovetoResultList($mis, n$)
27:     if $countOnes(mis.flag) = n$ then
28:         Calculate the ranking score $mis.score$
29:         if $mis.optimal = true$ then
30:             Append $mis$ to $OList$
31:         else
32:             Append $mis$ to $SList$
33:     else
34:         if $S = \emptyset$ then
35:             $mis_{top} = top(S)$
36:             Link $mis$ to $mis_{top}$
37:             $CopyFlags(mis_{top}, mis, n)$
38:         else
39:             Calculate the ranking score $mis.score$
40:             Append $mis$ to $PList$
41: procedure CopyFlags($parent, child, n$)
42:     if $countOnes(parent.flag) < n$ then
43:         $parent.optimal = false$
44:     $parent.flag = parent.flag | child.flag$
45:     $parent.num = parent.num + child.num$
```

followed by sub-optimal results, and finally partial match results. Within each class of results, the individual results are ordered by their ranking score.

As mentioned in the previous section, each result in a result list is a pointer that points to a list L of MISs (i.e. MIS subtree). To display that result, we need to recover the original nodes in these MISs and organize these nodes in the same

Algorithm 3 XSMis_Output(OList, SList, PList)

```
 1: for $r$ in $OList$ do
 2:     $MS = RetrieveNodes(r)$
 3:     $OutputResult(MS)$
 4: for $r$ in $SList$ do
 5:     $MS = RetrieveNodes(r)$
 6:     $OutputResult(MS)$
 7: for $r$ in $PList$ do
 8:     $MS = RetrieveNodes(r)$
 9:     $OutputResult(MS)$
10: procedure OutputResult($MS$)
11:     while $\neg isEnd(MS_1) \wedge ... \wedge \neg isEnd(MS_n)$ do
12:         $n_i = getMinNode(MS)$
13:         $Display(n.name, n.level)$
14:         $Advance(MS_i)$
```

way as they appear in the original data tree. To achieve this, we build a *region index* on the *start* attributes of the MISs. Each *start* value *val* in the leaf nodes points to the set of nodes in the MIS whose start value is *val*, and for each node, there is the label or value as well as the normal region code in the original data tree. This index records the nodes and the organization of these nodes within each MIS. Using this index, the procedure *RetrieveNodes* retrieves all the nodes that are in the list L of MISs. The retrieved xml nodes of each MIS are stored in the stream MS_i ($0 < i < N$, N is the number of MISs in L) and $MS = \{MS_1, \ldots, MS_N\}$. The procedure $getMinNode(MS)$ returns the xml node which has the smallest *start* value among all the streams MS_i in MS. Then this node can be displayed with an indent according to its *level* value (line 13). An example of the final output is shown in Figure 5 (c).

5. EXPERIMENTS

In this section, we present the experiment results on the efficiency and effectiveness of our approach against XRank [5], SLCA [13] and MaxMatch [11].

The approaches are evaluated with the following metrics: (1) processing time, (2) scalability on the document size, (3) effectiveness based on precision, recall and F-measure.

5.1 Experimental setup

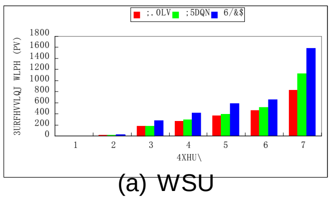
The xml document parser we used is the XmlTextReader Interface of Libxml2 [3]. The *keyword index* and *region index* are implemented in C++ and stored with Berkeley DB [2].

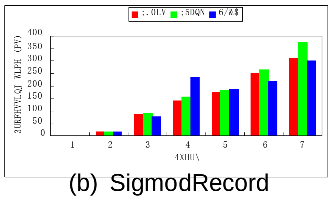
We implemented XKMis, XRank and SLCA in C++. The executable file of MaxMatch is provided by its author. All the experiments were performed on a 1.6GHz Intel Centrino Duo processor laptop with 1G RAM. The operating system is Windows XP. We used the data sets WSU, SigmodRecord and Mondial obtained from [1] for evaluation. In order to get a larger data size, we replicated these three data sets 20 times, and the sizes of them are 31M, 9M and 34M respectively.

We selected seven keyword queries for each data set. The queries are listed in Table 1. Besides the queries, the total frequency of keywords (i.e. the total number of occur-

Name	Dataset	Keyword Query	Total Frequency
QW1	WSU	ACCTG	640
QW2	WSU	CAC, 101	7,020
QW3	WSU	instructor, MCELDOWNEY	78,640
QW4	WSU	bldg, TODD, course, ECON	170,500
QW5	WSU	course, ACCTG, times, place	236,120
QW6	WSU	course, ECON, crs, title, days	315,420
QW7	WSU	course, crs, sect, title, credit, days, times, place	627,880
QS1	SigmodRecord	Karen, Anthony	220
QS2	SigmodRecord	Anthony, Data, Design	4,760
QS3	SigmodRecord	Stephen, Database	7,120
QS4	SigmodRecord	volume, 11, article	32,082
QS5	SigmodRecord	article, Data, System	36,660
QS6	SigmodRecord	database, author, Tom	81,861
QS7	SigmodRecord	issue, volume, article, title, initpage, endpage, author	197,902
QM1	Mondial	Europe	60
QM2	Mondial	Continent	4,680
QM3	Mondial	Tirane, population	88,540
QM4	Mondial	organization, name, members	111,182
QM5	Mondial	city, name, longitude, latitude	201,642
QM6	Mondial	country, name, population	465,352
QM7	Mondial	country, name, capital, population, datacode, government	504,642

Table 1: Keyword queries


(a) WSU


(b) SigmodRecord

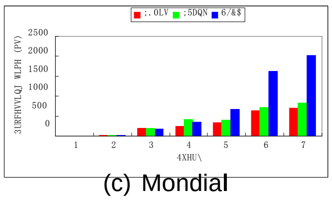
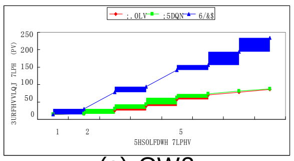

(c) Mondial

Figure 6: Processing time

rences of the keywords in the data tree) for each query is also listed in ascending order. This value is used to analyze how the performance changes when the frequency of a query increases.

5.2 Processing time

We compared the processing time of XKMis with XRank and SLCA, which have better efficiency than MaxMatch [11]. The comparison results are illustrated in Figure 6. As shown, XK-Mis outperforms XRank in all queries even though it needs to construct MIS subtrees and takes partial match results into consideration. Both XKMis and XRank are stack-based algorithms. The improvements on performance mainly come from the following. First, XKMis is not a LCA-based approach, so much time is saved on the computation of LCAs. Second, XKMis uses region-based coding. Compared with the Dewey coding, it will be more efficient to determine the ancestor-descendant relationship. In addition, our approach has much less entries pushed/popped-up into/from the stack. Given a keyword query, there is a great chance that several keywords appear in the same MIS, and our approach guarantees these same MISs are pushed into the stack only once. In contrast, each keyword match node in XRank may push serval entries into the stack. For example, suppose a keyword match node is coded with 0.1.1. Three entries (i.e. 0, 1, 1 with some other attributes) will be pushed into the stack. The more keywords appear in the same MIS, the more time will be saved on stack operations. For example, the query QW7 and QS7 have the most keywords among the queries and these keywords are very likely to appear in the same MIS. As shown in Figure 6, the improvement on performance of these two queries is greater than for other queries.


(a) QW3

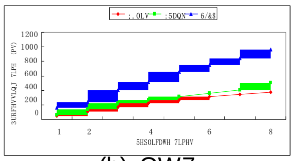
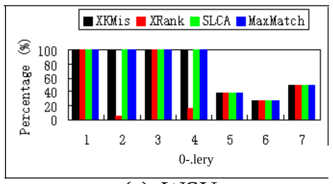

(b) QW7

Figure 7: Processing time with increasing document size

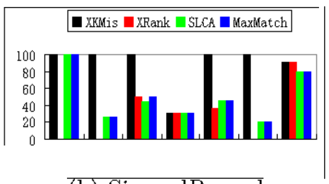
XKMis significantly outperforms SLCA in most queries even though some time is spent on constructing MIS subtrees. However, in some queries (e.g., QS3, QS6 and QS7), SLCA achieves better performance than both XKMis and XRank. This is mainly because of the *false negative* problem of SLCA. Some valid results are ignored by the algorithm.

5.3 Scalability

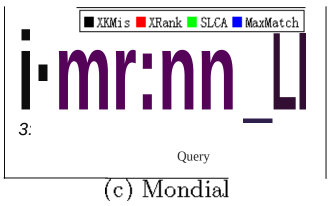
We compare the scalability of XKMis, XRank and SLCA using the queries over the data set WSU. In order to make the evaluation more accurate, we selected a short query QW3 which includes two keywords and a long query QW7 which has eight keywords. The parameter of scalability selected for evaluation is the document size. We replicated the WSU data set of size 1.6M between 1 and 8 times to get increasingly larger data sets. The processing time of query QW3 and QW7 over these data sets are shown in Figure 7. It can be seen that the processing time of these three approaches increases linearly when the document size increases. XKMis and XRank have the similar increasing speed. SLCA increases the fastest among the three approaches.



(a) WSU

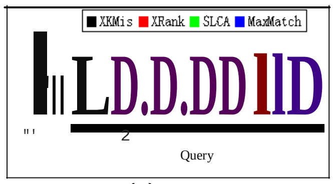


(b) SigmodRecord

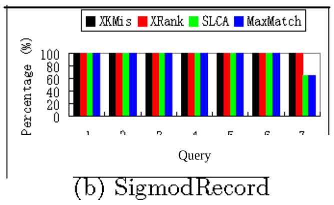


(c) Mondial

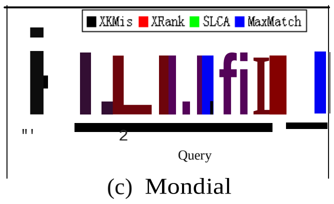
Figure 8: Precision



(a) WSU



(b) SigmodRecord



(c) Mondial

Figure 9: Recall

5.4 Effectiveness

We evaluate the effectiveness of XKMis, XRank, SLCA and Max-Match based on *precision, recall* and *F-measure.* These measures are extensively used in IR research to evaluate the relevancy of the results. Precision measures the percentage of retrieved results desired by users. Recall is the probability that a relevant result is retrieved by the query. F-measure is the weighted harmonic mean of precision and recall. They are defined as follows:

$$Precision = \frac{|Rel \cap Ret|}{|Ret|} \quad (4)$$

$$Recall = \frac{|Rel \cap Ret|}{|Rel|} \quad (5)$$

$$F = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (G)$$

Here, Rel is the set of *relevant* results that should be found (We transformed the keyword query to XQuery to get such results), Ret is the set of results actually retrieved using the keyword search system. For F-Measure, precision and recall are evenly weighted.

As shown in Figure 8, XKMis generally achieves higher precision than XRank, SLCA and MaxMatch. However, the precisions of some queries are not high (e.g., QW5, QW6, QW7, QS4 and QM3). This is mainly because the user is not interested in all the information contained in a MIS. For example, for the query QM3, the user is only interested in the population of the city Tirane, but our approach also returns some other information of Tirane. Actually, this kind of results is acceptable. First, the size of a MIS is not big, so it will not bring difficulties on finding the desired information with the help of highlight. Second, the extra information is still about the same MIS, not others, so it is understandable and will not bring confusion to users. The precision of the query QM6 is not high either. It is because the keywords "country", "name" and "population" widely exist in the MISs of city as well as the MISs of country. However, these MISs of city are not desirable. Actually, this kind of problem can be properly solved in our approach. We can further classify the results according to their MIS-types. The users should know which MIS-type they really want. In this case, the user wants the MISs of country, so he can choose the MISs of country to be shown by the system. The MISs of city are effectively filtered out. We leave this as part of our future work. The precisions of many results returned by XRank are very low (e.g., QW2, QW4, QSl, QS2 and QS5). This is mainly because large trees may be returned due to the false positive problem. For example, for the query QSl, two articles in different issues written by Karen and Anthony respectively are connected via the LCA SigmodRecord. This causes a very large tree rooted at SigmodRecord to be returned to users. This is unacceptable even though the keywords are highlighted because it is very difficult for users to find their desired results. The precisions of some results returned by SLCA and MaxMatch are also low because of the false positive problem. But they are better than XRank, because the semantics of SLCA and MaxMatch prevent some false positive problems. However, for the query QS2, QS3 and QS5, the false positive problems can not be avoided.

F-Measure	XKMis	XRank	SLCA	MaxMatch
WSU	0.80	0.43	0.67	0.67
SigmodRecord	0.87	0.44	0.66	0.68
Mondial	0.79	0.51	0.48	0.48

Table 2: Comparison on F-Measure

As shown in Figure 9, XKMis also achieves higher recall than XRank, SLCA and MaxMatch especially when a query has only one keyword. For XRank, SLCA and MaxMatch, if a query includes only one keyword, they just return the nodes which contain that keyword. This kind of results is not informative and not desirable. The recall of SLCA and MaxMatch is worse than XRank because some valid results are ignored due to the false negative problem. Actually, in some queries, it is trivial for XRank and SLCA to achieve high recall because they return the entire XML data tree to users.

We calculated the average F-measure of the queries over each data set and they are listed in Table. 2. It can be seen that XKMis achieves higher F-measure than both XRank, SLCA and MaxMatch.

6. CONCLUSION

We presented our XML keyword search system XKMis. Unlike previous work, our method is not based on the lowest common ancestor (LCA) or its variant. Instead, we divide the xml nodes into minimal information segments (MISs) and return MIS-subtrees which consist of MISs that are more logically connected by the keywords. We conducted extensive experiments to compare our approach with XRank and SLCA. The better overall performance, scalability and search quality have been verified by our experiments.

7. REFERENCES

[1] http://www.cs.washington.edu/research/xmldatasets.

[2] http://www.oracle.com/database/berkeley-db/.

[3] http://xmlsoft.org/.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

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

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

[7] 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.

[8] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over XML documents. In *CIKM*, pages 31–40, 2007.

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

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

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

[12] J. Xu, J. Lu, W. Wang, and B. Shi. Effective keyword search in XML documents based on MIU. In *DASFAA*, pages 702–716, 2006.

[13] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in XML databases. In *SIGMOD Conference*, pages 527–538, 2005.