Schema-Free XQuery

Yunyao Li* **Cong Yu*** **H. V. Jagadish***

Department of EECS, University of Michigan
Ann Arbor, MI 48109, USA
{yunyaol, congy, jag}@eecs.umich.edu

Abstract

The widespread adoption of XML holds out the promise that document structure can be exploited to specify precise database queries. However, the user may have only a limited knowledge of the XML structure, and hence may be unable to produce a correct XQuery, especially in the context of a heterogeneous information collection. The default is to use keyword-based search and we are all too familiar with how difficult it is to obtain precise answers by these means. We seek to address these problems by introducing the notion of Meaningful Lowest Common Ancestor Structure (MLCAS) for finding related nodes within an XML document. By automatically computing MLCAS and expanding ambiguous tag names, we add new functionality to XQuery and enable users to take full advantage of XQuery in querying XML data precisely and efficiently without requiring (perfect) knowledge of the document structure. Such a Schema-Free XQuery is potentially of value not just to casual users with partial knowledge of schema, but also to experts working in a data integration or data evolution context. In such a context, a schema-free query, once written, can be applied universally to multiple data sources that supply similar content under different schemas, and applied "forever" as these schemas evolve. Our experimental evaluation found that it was possible to express a wide variety of queries in a schema-free manner and have them return correct results over a broad diversity of schemas. Furthermore, the evaluation of a schema-free query is not expensive using a novel stack-based algorithm we develop for computing MLCAS: from 1 to 4 times the execution time of an equivalent schema-aware query.

* Supported in part by NSF under grant number IIS 0219513, by NIH under grant number LM08106-01, and by a gift from Microsoft.

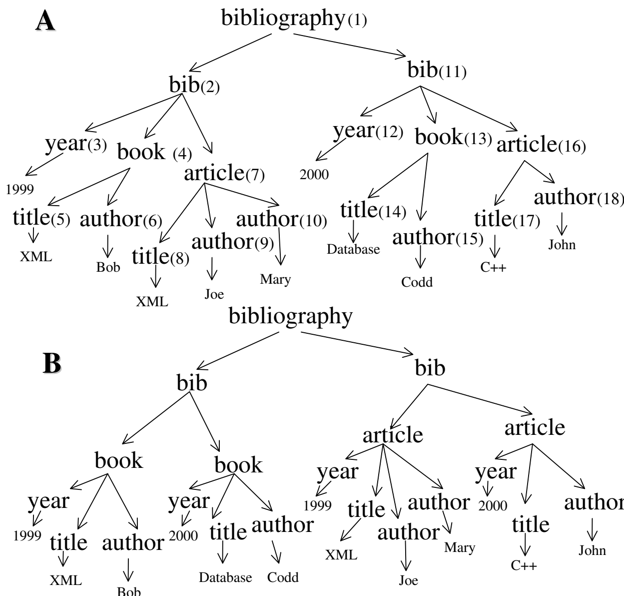
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

XML is gradually becoming the standard in exchanging and representing data. Not surprisingly, effective and efficient querying of XML data has become an increasingly important issue. Traditionally, research work in this area has been following one of the two paths: the structured query approach and the keyword-based approach. XQuery [9] is the generally acknowledged standard of the former, while the latter class has several recent suggestions, including XKeyword [17] and XSEarch [11]. Both approaches have their advantages and disadvantages. Fully structured query (e.g., XQuery) works effectively with the structure, can convey complex semantic meaning in the query, and therefore can retrieve precisely the desired results. However, if the user does not know the (full) structure, it is difficult to write the right query. Even if the user does know the schemas, when data is to be amalgamated from multiple sources with different schemas, it typically will not be possible to write a single query applicable to all sources; rather, multiple queries will have to be written (or at least generated through translation), a process that is complex and error-prone. Keyword-based query can overcome the problems with unknown schema or multiple schemas because knowledge of structure is not required for the query. However, this absence of structure leads to two serious drawbacks. First, it is often difficult and sometimes impossible to convey semantic knowledge in pure keyword queries. Second, the user cannot specify exactly how much of the database should be included in the result.

Consider the example in Figure 1 showing the same bibliography data arranged in two different formats: A organizes publications based on the year of publication and B organizes publications according to their type (*book* or *article*). Let's first look at Query 1, which is a simple query asking for some information (*title* and *year*) on a publication given a certain condition (*author* is "Mary"). To construct an XQuery to represent this simple query, the user faces two challenges: first, she has to know that "publication" in the schema is actually presented as *book* and *article* in both schemas; second, she has to know that *title* and *author* are the child elements of "publication", while *year* could be either a



Query 1: Find title and year of the publications, of which Mary is an author.
Query 2: Find additional author of the publications, of which Mary is an author.
Query 3: Find year and author of the publications with similar titles to a publication of which Mary is an author.

Figure 1: Querying XML data with multiple schemas.

child or a sibling. Depending on the schema, the resulting XQuery expression is non-trivial even for this simple query in this very small example. The keyword based approach, on the other hand, often returns results that include too many irrelevant answers. Query 1 (with the underlined keywords) on data in A may return the *bib* node **2**, which contains not just the desired *article* node **7**, but also the unwanted *book* node **4**. Queries 2 and 3 pose even greater challenges for the keyword-based approach. Keywords cannot distinguish the two different *author*s in Query 2 and will simply return node **10** whose content is "Mary". Query 3 involves two logical structures linked together through a value join–it even shares the same set of keywords with Query 1! Therefore it is hard to imagine how the limited semantic capacity of a keyword search specification could capture user intent.

In this paper, we developed a framework that enables users to query XML data exploiting whatever partial knowledge of the schema they have. If they know the full schema, they can write regular XQuery. If they do not know the schema at all, they can just specify keywords. Most importantly, they can be somewhere in between, in which case the system will respect whatever specifications are given.

The notion of Lowest Common Ancestor (LCA) (of individual term/tag matches) has been suggested (e.g. Meet [22]) as an effective mechanism to identify segments of the database of interest to a pure keyword query. While this intuition is reasonable, we show that LCA can frequently be too inclusive. We refine LCA and define the concept of a *Meaningful Lowest Common Ancestor Structure* (MLCAS). Each MLCAS is an XML fragment that meaningfully relates together the nodes corresponding to the relevant variables in the XQuery expression. In Section 2, we show how this structure, and its root node, can be referenced and manipulated in an XQuery expression with embedded **mlcas** functions.

An **mlcas** function precisely specifies a particular non-trivial computation. We may prefer to hide this from a novice user. Furthermore, users may lack knowledge not only about the structure, but also about the specific tag names in the structure. In Section 3, we propose Schema-Free XQuery to address these two issues. Ordinary users are thus able to write simplistic XQuery expressions, specifying keywords and/or tag names and/or structural restrictions, ranging all the way from an open-ended IR-style keyword specification to a completely specified full-fledged XQuery expression.

MLCAS computation is a core part of Schema-Free XQuery evaluation. In Section 4, we show how to accomplish this using standard XQuery evaluation operators. We then introduce a novel stack-based algorithm to compute MLCAS more efficiently, in a manner reminiscent of containment join.

In Section 5, we present an experimental evaluation of our proposal, in terms of both the quality of the results produced and the time taken to produce them. Over both XMark, a standard XML Benchmark, and a wide variety of autonomously created schemas in a well-circumscribed domain (publication lists) we found that Schema-Free XQuery almost always produced exactly the desired results. Moreover, the time taken to do so was only somewhat greater than an equivalent schema-specific query would require.

Finally, we discuss related work in Section 6 and conclude in Section 7.

2 MLCAS

In this section, we describe the concept of MLCAS and present the **mlcas** function as an addition to standard XQuery. The resulting **mlcas**-embedded XQuery gives a user the full expressive power of XQuery while forgiving incompleteness in schema specification.

We begin first with a description of the XML data model that we employ. An XML document is a rooted, ordered, and labeled tree. Nodes in this rooted tree correspond to elements in the XML document.

Definition 2.1 (Descendant-Or-Self) *Tree node n_d is said to have a descendant-or-self relationship with n_a if it is a descendant of n_a or is equal to n_a, denoted as descendant-or-self(n_d, n_a) = true.*

Definition 2.2 (LCA) *Let the set of nodes in an XML document be N. For $d_1, d_2 \in N$, $a \in N$ is the LCA of d_1 and d_2 if and only if:*

- *descendant-or-self(d_1, a) = true, and*
- *descendant-or-self(d_2, a) = true, and*
- *$\forall\ a' \in N$, if descendant-or-self(d_1, a') = true and descendant-or-self(d_2, a') = true, then descendant-or-self(a, a') = true.*

a is denoted as LCA(d_1, d_2).

2.1 Motivation for MLCAS

An XML query typically involves one or more sets of structurally related XML elements that are the processing context used by the query (either to evaluate conditions or to return results). If a user knows the document structure, she can write a meaningful query in XQuery specifying exactly how the nodes involved in the query are structurally related with each other. Without knowledge of the structural relationships, as long as the user knows the element tag names, she can still write an XQuery specifying only the tag names of elements involved in the query. Figure 2 shows one such expansion for Query 1 in Figure 1. A literal evaluation of this expansion will retrieve many meaningless results because the default context is too general (i.e., all of *bib.xml*).

Given the structured nature of XML, it is natural to find the LCA of the set of nodes specified, and treat the subtree rooted at this node as the context for query evaluation. In fact, this idea has been employed in several previously proposed systems [11, 17] and works well in certain cases. For example, consider nodes **8** (*title*) and **10** (*author*) in Figure 1. The LCA of these two nodes is node **7** (*article*) and the subtree rooted at node **7** does make a good context: the *title*, *author*, and *article* nodes form a logical entity together. However, blindly computing the LCA can bring together unrelated nodes. For example, consider a different pair of nodes in Figure 1: nodes **5** (*title*) and **10** (*author*). Their LCA is node **2** (*bib*), whose subtree contains many *book*s and *article*s and is clearly not an appropriate context for the query evaluation. We address this problem by introducing the notion of MLCAS, and using it as the refined context for query evaluation.

2.2 MLCA

A node in an XML document, along with its entire subtree, typically represents a real-world entity. The tag name usually identifies the type of the entity (called *entity type* to distinguish it from the data type used by XML Schema [25]).

Definition 2.3 (ENTITY TYPE) *An entity type (or simply type) of a node n in an XML tree is defined as the tag name (label) of n. Two nodes n_1 and n_2 are of the same entity type $\mathcal{T}$ if and only if they have the same tag name.*

In the presence of ontology (i.e. type hierarchy), nodes with different tag names may still be regarded as of the same type. For example, *book* and *article* nodes can be deemed as of the same super-type $\mathcal{P}$ (*Publication*). For the simplicity of presentation, we do not consider ontology-guided type matching here.

```
for    $a in doc("bib.xml")//author,
       $b in doc("bib.xml")//title,
       $c in doc("bib.xml")//year
where  $a/text() = "Mary"
return <result> { $b, $c } </result>
```

Figure 2: Query 1 in XQuery within no structural knowledge

We now describe, through the diagrams in Figure 3, what it means intuitively when we say two nodes are meaningfully related to each other. Let node n_1 represent an entity of type $\mathcal{A}$, and node n_2 represent an entity of type $\mathcal{B}$. First, suppose that n_1 is an ancestor node of n_2 (shown in Figure 3(a)), we believe n_1 and n_2 are meaningfully related to each other. Second, consider the situation where two nodes have no hierarchical relationship with each other. Suppose the LCA of n_1 and n_2 is n (shown in Figure 3(b)), we can regard both entities represented by n_1 and n_2, respectively, belong to the entity represented by n. Therefore, nodes n_1 and n_2, regardless of their types, are related to each other by belonging to the same entity represented by n, which is regarded as the *Meaningful Lowest Common Ancestor*(MLCA) of n_1 and n_2. However, there is an exception to this second case. As demonstrated by Figure 3(c), let there be a node n'_2 of the same type as node n_2, and the LCA of n_1 and n'_2 be n'. If n is an ancestor node of n', we should then conclude that nodes n_1 and n_2 are not meaningfully related to each other because node n'_2, which is of the same type as n_2, is more related to n_1 under the node n', which is actually the MLCA of n'_2 and n_1.

Consider the previously mentioned example of nodes **5** and **10** in Figure 1, their LCA is node **2**. However, it is not their MLCA, because it is an ancestor of node **7**, which is an MLCA of nodes **8** and **10**, and node **8** is of the same type as node **5** (both are *title*s). In fact, the entities *title* and *article* are related to each other by belonging to the same "publication" (*book* or *article*). Nodes **5** and **10** are not related (i.e., not in the same MLCAS) because they belong to different publications.

We now formalize this idea. First of all, given two sets of nodes, where nodes within each set are of the same type, we define how to find pairs of nodes that are meaningfully related to each other from these two sets.

Definition 2.4 (MLCA of two nodes) *Let the set of nodes in an XML document be N. Given A,B $\subseteq$ N, where A is comprised of nodes of type $\mathcal{A}$, and B is comprised of nodes of type $\mathcal{B}$, the Meaningful Lowest Common Ancestors Set C $\subseteq$ N of A and B satisfies the following conditions:*

- *$\forall\ c_k \in C,\ \exists\ a_i \in A,\ b_j \in B$, such that $c_k = LCA(a_i,b_j)$. c_k is denoted as $MLCA(a_i, b_j)$.*
- *$\forall\ a_i \in A,\ b_j \in B$, if $d_{ij} = LCA(a_i,b_j)$ and $d_{ij} \notin C$, then $\exists\ c_k \in C$, $descendant(c_k, d_{ij}) = true$.*

The set C is denoted as MLCASET(A, B).

A pair of nodes (a, b), where a is of type $\mathcal{A}$ in set A and b is of type $\mathcal{B}$ in set B, are regarded as meaningfully related to each other if and only if c, the LCA of a and

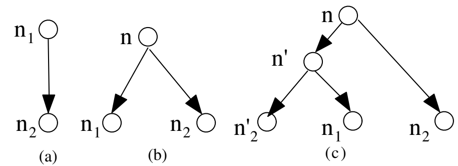


Figure 3: Structural relationships among nodes

b, belongs to C, where C is MLCASET(A, B). This restriction ensures that only the most specific results are returned. If an element's subelement is returned, then the element would not be returned, because its subelement has a closer relationship between the entities represented by nodes in A and B respectively. Given multiple sets of nodes, where nodes within each set are of the same type, we can easily extend Definition 2.4 to define the MLCA of multiple nodes:

Definition 2.5 (MLCA of multiple nodes) *Let the set of nodes in an XML document be N. Given A_1, A_2, ..., $A_m \subseteq N$, where $\forall$ j, $a_{ij} \in A_i$ is of type $\mathcal{A}_i$ ($i \in [1,\ldots,m]$), a Meaningful Lowest Common Ancestor c = MLCA($a_1,\ldots,a_m$), where $a_i \in A_i$ ($i \in [1,\ldots,m]$), satisfies the following conditions:*

- *$\forall j,k \in [1,\ldots,m]$ ($j \neq k$), $\exists$ m = MLCA(a_j, a_k), $m \neq$ `null` and descendant-or-self(m,c) = true.*
- *$\exists j,k \in [1,\ldots,m]$ ($j \neq k$), c = MLCA(a_j, a_k).*

2.3 MLCAS

We have seen above how to find MLCA for multiple nodes. However, this in itself is not enough since the same node could be the meaningful lowest common ancestor to many different sets of nodes. For instance, given a *book* with two *author*s, the same *book* node can be the MLCA for the *title* node and each of the *author* nodes, separately. Consider the query in Figure 2 against the data in schema A in Figure 1. Simply computing the MLCA of nodes (*author*,*title*,*year*) involved in the query will regard the subtrees rooted at nodes **2** and **11** as the context for query evaluation. Although they do contain the desired result, they often include too much irrelevant information. A user must read the results returned and manually discover the desired answer. This could require a significant amount of work in a large database. Even worse, the system may return additional (incorrect) answers. In this particular example, the user requests the nodes *year* and *title*, answers (**3**,**5**) and (**3**,**8**) will be returned. The former is a wrong answer because only the latter *title* is the desired result. We resolve this ambiguity by identifying not just the MLCA itself, but rather an entire structure, MLCAS, for each such established relationship.

Definition 2.6 (MLCAS) *Let the set of nodes in an XML document be N. Given A_1, A_2, ..., $A_m \subseteq N$, where $\forall$ i, $a_{ij} \in A_j$ is of type $\mathcal{A}_j$ ($j \in [1,\ldots,m]$), the Meaningful Lowest Common Ancestor Structure Set S = $\{(r, a_1,\ldots,a_m) \mid r \in N, a_i \in A_i$ ($i \in [1, \ldots, m]$), r = MLCA($a_1,\ldots,a_m$) $\}$. Each element of this set is denoted as MLCAS($a_1,\ldots,a_m$), with r as its root.*

Each MLCAS is a refined context for query evaluation, and contains only the nodes that are meaningfully related to one another. If an MLCAS satisfies the search conditions, it is unlikely to contain a wrong answer. For example, for the running example Query 1 in Figure 1, expressed as shown in Figure 2, we obtain several MLCASs, including (**2**,**10**,**8**,**3**) and (**11**,**15**,**14**,**12**). The only MLCAS satisfying the original search condition *\$a/text()="Mary"* is (**2**,**10**,**8**,**3**). Hence,the result is (*title*="XML",*year*="1999"), which is exactly the desired result.

Finally, we would like to point out the differences between the concept of MLCAS and the concept of interconnected nodes employed by the XSEarch system [11]. Both concepts are designed to capture the meaningful substructure of the XML document based on both the tag names and the keywords provided in a query. Interconnected nodes are the set of connected nodes with a root node, where no two internal nodes are of the same type (i.e., having the same tag name) and the root node is the LCA of leaf nodes. This concept works well for simple XML data where logically equivalent entities always have the same tag name. However, it does not recognize meaningful structure when those entities (e.g., *book* and *article* in the previous example) have different tag names. In addition, it does not work well on XML data with more than one logical hierarchy. Consider evaluating running example Query 1 against the data in Figure 4, the unrelated *title:Streaming* will be returned by XSEarch. Due to the fact that no two nodes have the same tag name along the path, XSEarch fails to recognize that this title is actually more meaningfully associated with *author:John* under *ref.* Search based on MLCAS, on the other hand, can easily recognize this fact and therefore avoid returning the incorrect result.

2.4 Adding mlcas Function to XQuery

In this section, we introduce a new language construct, **mlcas** function, to the standard XQuery language:

Definition 2.7 (mlcas Function) *mlcas($a_1,\ldots,a_n$)is a function that returns (i) root node of MLCAS(a_1, ...,a_n), if it exists, (ii)* `null` *otherwise.*

Figure 5 shows how each of the three running queries presented in Figure 1 can be expressed in the XQuery enriched with the **mlcas** function. Each query will retrieve precisely the desired result, when executed against either example schema in the figure.

Query 1 is the most straightforward. Given the tag names of individual nodes, the condition exists *mlcas*(\$a, \$b, \$c) defines the context for evaluation to be the MLCAS of those nodes and filters out any node that cannot be part of any MLCAS. The query is flexible since it does not require user to know the exact

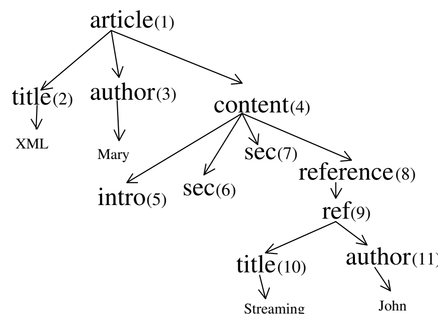


Figure 4: XML data with multiple hierarchies

Query 1:

```
for $a in doc("bib.xml")//author,
    $b in doc("bib.xml")//title,
    $c in doc("bib.xml")//year
where  $a/text() = "Mary"
       and exists mlcas($a,$b,$c)
return  <result> {$b, $c} </result>
```

Query 2:

```
for $a in doc("bib.xml")//author,
    $b in doc("bib.xml")//author,
where  $a/text() = "Mary" and $a != $b
       and exists mlcas($a,$b)
return  $b
```

Query 3:

```
for $y in doc("bib.xml")//year,
    $a1 in doc("bib.xml")//author,
    $t1 in doc("bib.xml")//title,
    $t2 in  {
      for $a in doc("bib.xml")//author,
          $t in doc("doc.xml")//title
      where  $a/text() = "Mary"
             and exists mlcas($a,$t)
      return  $t }
let  $m := mlcas($y,$a1,$t1)
where  $t1 ≈ $t2 and exists $m
return  <result> {$y, $a1} </result>
```

Figure 5: Example XQueries with **mlcas** function

relationships between nodes of the three types. Query 2 shows another aspect of the flexibility in the **mlcas** function: the individual nodes do not have to be of different types. By combining the conditions $\$a != \$b$ and exists $mlcas(\$a, \$b)$, the only MLCASs retained are publications with at least two different *author*s. Query 3 shows a more complex example. It contains two contexts for evaluation: one in the outer query, which contains *year*, *author*, and *title*; the other in the inner query, which contains *author* and *title*. The two contexts are linked together through the similarity join $\$t1 \approx \$t2$. This query is difficult to express in any keyword based approach simply because the keyword to be used to match the content of title is only known during the runtime. Although the binding of the result of **mlcas** function to a variable $\$m$ is not necessary, it is shown here to illustrate that the root of the MLCAS can be manipulated just like any other regular elements in the XML document. If we evaluate this query against data in schema A of Figure 1, the only MLCAS satisfying the conditions in the inner query will be (**7**,**10**,**8**), and the only *title* to be returned is "XML". The outer query, without considering the similarity join, will have several MLCASs, including (**2**,**3**,**6**,**5**), (**2**,**3**,**9**,**8**), (**2**,**3**,**10**,**8**), (**11**,**12**,**15**,**14**), and (**11**,**12**,**18**,**17**). Only the first three have *title* similar to "XML" and are the final MLCASs when we consider the similarity join. The final results to be returned are therefore (*year* = "1999", *author* = "Bob"), (*year* = "1999", *author* = "Joe"), and (*year* = "1999", *author* = "Mary").

3 Schema-Free XQuery

To achieve true flexibility in the query (i.e., Schema-Free XQuery), one first needs to address the issue of *structure ambiguity*, where the relationship among elements is unclear. While the use of **mlcas** function inside XQuery, as described in Section 2.4, effectively deals with the *structure ambiguity*, adding one more language construct to the already complex XQuery will likely prevent ordinary users from adopting it: we would like to allow users to write XQuery using the standard syntax as much as possible and have the system automatically figure out what to do. We present our solution to this through the use of mlcas keyword in Section 3.1. The second issue is *tag name ambiguity*, where the exact tag name of a particular element is unknown (although the user should have a rough idea what the tag name is in general) and we address this issue in Section 3.2. As we will show in this section, a Schema-Free XQuery is underspecified: our task is to derive a completely specified query that best captures the user's intent.

3.1 MLCAS Transformation

Toward the goal of allowing user to take advantage of **mlcas**-embedded XQuery while maintaining the simplicity of XQuery, we propose adding a simple mlcas keyword to the standard XQuery. The keyword is used to ask the system to transform the original simplistic XQuery into an XQuery with **mlcas** function, which then resolves the structure ambiguity automatically. The following query illustrates a simple mlcas-enhanced XQuery representing Query 1 in Figure 1.

```
for $a in mlcas doc("bib.xml")//author,
    $b in mlcas doc("bib.xml")//title,
    $c in mlcas doc("bib.xml")//year
where  $a/text() = "Mary"
return  <result> { $b, $c } </result>
```

This query can be automatically transformed into Query 1 in Figure 5 through a simple transformation algorithm, which will be briefly discussed later. One restriction we put on the semantics of the mlcas-enhanced XQuery is to have all mlcas marked variables within one FLWOR block belonging to the same MLCAS: we believe this is the intention for most queries and it simplifies the construction of the query for the user. However, nested queries will have separate MLCASs. For example:

```
for $y in mlcas doc("bib.xml")//year,
    $a1 in mlcas doc("bib.xml"//author,
    $t1 in mlcas doc("bib.xml")//title,
    $t2 in {
        for $a in mlcas doc("bib.xml")//author,
            $t in mlcas doc("bib.xml")//title
        where  $a/text() = "Mary"
        return  $t
    }
where  $t1 ≈ $t2
return  <result>{$y, $a1}</result>
```

As shown in this example, the MLCAS is designed to have a scope that is local to the query. Hence, the MLCAS formed from the mlcas marked variables in the

subquery is different from the MLCAS formed from the mlcas marked variables in the parent query. The two MLCASs are linked together by a similarity value join and our system will transform this query into Query 3 in Figure 5. Furthermore, the mlcas marked variables do not always have to represent descendant elements with respect to the document root. If the user has a better understanding of the document structure, she can explicitly specify the part that she knows and leaves the part that she doesn't know to the system. Consider the query:

```
for    $r in doc("bib.xml")//bib[1],
       $a in mlcas $r//author,
       $b in mlcas $r//author
where  $a/text() = "Mary" and $a != $b
return $b
```

The user here explicitly wants the two *author*s to be within the first *bib* element, she does so by associating the first *bib* element with the variable $r, and marking the relationship between $r and the two *author*s with mlcas. The system will then take all *author*s that are descendants of the first *bib* element and try to compute MLCASs from those nodes only. The transformed query is not shown, and is similar to Query 2 in Figure 5: it is slightly different on account of the additional $r binding.

Transformation algorithm: Here we describe the algorithm that accomplishes the above transformations in a brief outline, not the complete presentation, due to the space limitation. The algorithm begins by taking an arbitrary expression (XQuery, XPath, binary condition, etc.) as its input. If the expression does not have any XQuery (i.e. FLWOR block) inside, it is simply returned as it is. Otherwise, the algorithm extracts all mlcas marked variables within the XQuery at the current nesting level into a **mlcas** function in the where clause of the **mlcas**-embedded XQuery to be returned. The procedure repeats for each nested XQuery it recursively extracts from the input expression. As a result, each nested query with mlcas keyword will have one and at most one single **mlcas** condition, which is consistent with our intention that all mlcas marked variables in a single query block belong to the same MLCAS.

3.2 Term Expansion

While mlcas-enhanced XQuery addresses the issue of structure ambiguity, it still relies on the correctness of element tag names in a given query. For example, in the queries shown in Section 3.1, if the document being queried upon uses *au* instead of *author* to denote the concept of author, none of the queries will be able to generate the correct results. In an *ad hoc* information retrieval task, a casual user is as unlikely to have perfect knowledge of those tag names as to have the perfect knowledge of the structure relationships. We call this issue *tag name ambiguity*: the discrepancy between a query term and its actual tag name counterpart in the document. According to [14], less than 20% of people choose the same term for a single well-known object. Although the statistics with regard to the tag name usage in XML data is not available, we expect the same issue will be common. In fact, in the real data we collected from the web, people use different names–"paper", "publication", "pub"–to mean the same concept. Apparently, mlcas-enhanced XQuery is still not schema-free. To resolve this problem, we propose to add a simple function expand to standard XQuery, indicating the user's lack of knowledge of the exact tag name. The system will then expand the particular tag name to match its equivalents in the XML document based on a domain-specific thesaurus[1]. For example,

```
for    $a in mlcas doc("bib.xml")//expand(author),
       $b in mlcas doc("bib.xml")//title
where  $a/text() = "Mary"
return $b
```

The tag name *author* in the query is indicated by the expand function as not exact, and can be matched to *au* by the system if *au* is recognized as the synonym of *author* based on the domain-specific thesaurus. The tag name *title*, however, is not marked (the user is sure of the exact spelling) and will not be expanded. This reflects the principle of Schema-Free XQuery: helping the user construct meaningful query when the knowledge of schema (in terms of both structure and tag name ambiguity) is missing, while giving the user power to express the exact meaning when the knowledge of the schema is present. In addition to domain-specific synonyms, an ontology-driven hierarchical thesaurus can be applied. For example, in Figure 1, both *book* and *article* can be regarded as a kind of *publication*. Therefore, a query tag name of *publication* can be expanded to match both *book* and *article* even though they are not considered as the same concept. Incorporating this ontology-driven term expansion into our framework raises some interesting issues (e.g., how to efficiently determine one term is contained in another) and is the subject of our future work. In the next paragraph, we describe our approach of implementing domain-specific synonym expansion.

Given the thesaurus, a naive implementation of term expansion is to issue multiple queries, each with the to-be-expanded tag name replaced by one of its synonyms in the thesaurus. However, the time cost is proportional to the total number of synonyms of all the expand marked tag names in the query. This is very expensive especially when the query evaluation cost is high. Here, we propose a more efficient approach using term normalization. For each set of synonyms within the thesaurus, one of them is designated as the standard (or normalized) form. When building tag name index on the XML document, two indices are built: one is the regular tag name index with the tag name as it is in the document as the key; the other is the normalized tag name index, where only the normalized form is used as the key. Whenever an element with a non-standard tag name is to be added to the normalized index, the standard tag name is fetched and the element is added to the position keyed by the standard tag name. At query time, if a tag name marked with the

[1] If the actual schema for the document(s) is available, the thesaurus can be derived from the actual schema. Otherwise, such a domain-specific thesaurus can be developed either by domain experts or through some standard information retrieval techniques like bootstrapping. In the worst case, a universal thesaurus like WordNet [2] can be used.

```
StackNode {
    NodeType    head; //an input node
    int maxID ;
    ListNode   EList [m];// m: total number of input lists
    bitset  relBits [m];}

ListNode {
    NodeType    node;
    int min ;}
```

Figure 6: Data structure of stack node

expand function in the query is non-standard, the standard name will be fetched and used as the key to the normalized tag name index. Using this approach, with some space overhead of storing the normalized tag name index (the index is built only once when the document is loaded and updated independent of queries, therefore its time cost has no impact on the query time), only one user query (with all expand marked tag names normalized) needs to be evaluated, the result is a faster query response time. We note that term expansion works only when query terms that are semantically close to the tag names in the XML data are provided: we expect this to be a reasonable assumption.

3.3 Summary

Marking structurally ambiguous elements with mlcas keyword and ambiguous tag names with expand function enables a user to query XML documents without perfect knowledge of either the structural relationship among the elements or their tag names. XQuery equipped with these two features has effectively become schema-free: the user only needs minimal knowledge of the schema to issue a query that is far more meaningful than a keyword query and far more flexible than the standard XQuery.

4 Computing MLCAS

MLCAS computation is central to Schema-Free XQuery evaluation. In Section 4.1, we show how MLCAS can be evaluated as a composition of standard access methods likely to be available in most XQuery engines. In Section 4.2, we present a more efficient algorithm for computing MLCAS directly.

In the ensuing discussion, for a Schema-Free XQuery with an embedded function **mlcas**(e_1, e_2,..., e_m), where e_i are the elements involved in the MLCAS, we use $IList[i] = \{a_{11}, a_{12}, \ldots, a_{1n_i}\} \subseteq N$, to represent a list of nodes matching e_i ($1 \leq i \leq m$) in the XML data.

4.1 Basic Implementation

Computing MLCAS can easily be implemented using the existing query standard operators. The basic idea is to find all the ancestors for each node in the *IList*s, and join nodes sharing common ancestors into trees such that the "leaf level" contains exactly one node from each *IList*, and each leaf node has *descendant-or-self* relationship with the root. For any pair of trees, we eliminate the one whose root is an ancestor (in the database tree) of the root of the other, as it conflicts with the definition of MLCAS (Definition 2.6). The remaining trees are returned as the MLCASs.

Theorem 4.1 *The time complexity of the straightforward implementation is $O(h^m \prod_{i=1}^{m} n_i)$, where h is the height of the XML data tree.*

```
MLCAS (I1, I2, ..., Im):
0.let the set of input nodes from I1, I2, ..., Im be I
1.while (unprocessed input or stack is not empty)
2.     let tmin(from Iindex)be the node with smallest StartPos in I
3.     while (stack is not empty &&
4.          tmin is not a descendant of current stack top)
5.        /* pop the top element in the stack */
6.        popped = stack→Pop(), top = stack→Top()
7.        if (popped and its Elists contain MLCASs)
8.           output popped /*no more MLCAS on current stack*/
9.           while (stack is not Empty) stack→Pop()
10.       /* popped will not be a root of any MLCAS*/
11.       else if (popped→head is a child of top→head)
12.          mark all the non-empty Elists of popped as Related
13.          /* if popped qualified to be part of an MLCAS */
14.          if (for any i, popped→Elist[i] or top→Elist[i] is empty)
15.             top→AppendLists(popped→GetLists()
16.          else if(for any i,j(i≠j),if top→ElistsRelated(i,j)=true
17.                then popped→ElistsRelated(i,j) = true)
18.             if(exists i,j(i≠j) that top→ElistsRelated(i,j)=false
19.                && popped→ElistsRelated(i,j)=true)
20.                /*delete nodes unqualified to be in an MLCAS*/
21.                delete all nodes from top→Elist[i], top→Elist[j]
22.             top→AppendLists(popped→GetLists())
23.       else let pt = popped→head→GetParent()
24.          mark all the non-empty Elists of popped as Related
25.          popped→ReplaceHead(pt) /*replace with parent*/
26.          stack→Push(popped)
27.    if (stack is empty)
28.       stack→Push(tmin), top = stack→Top()
29.       top→SetMaxID(0)
30.       /*set min of newnode be 0*/
31.       newnode=NewListNode(tmin, 0)
32.       top→Elist[index]→AppendNode(newnode)
33.    else
34.       oldtop = stack→Top(),stack→Push(tmin)
35.       top = stack→Top()
36.       top→SetMaxID(oldtop→GetMaxID())
37.       /*assign min to distinguish nodes of different MLCASs*/
38.       if (oldtop→Elist[index] is empty)
39.          newnode=NewListNode(tmin, oldtop→GetMaxID()
40.       else if(oldtop→Elist[index] not Related with other Elists)
41.             newnode=NewListNode(tmin,oldtop→GetMaxID())
42.          else
43.             top→SetMaxID(oldtop→GetMaxID()+1)
44.             newnode=NewListNode(tmin,top→GetMaxID())
45.       top→Elist[index]→AppendNode(newnode)
46.    read I for the next tmin
```

Figure 7: Algorithm MLCAS: it finds all MLCASs for the input nodes, and returns the root node for each MLCAS. Each input list I_k ($1 \leq k \leq m$) is a set of nodes of the same entity type, sorted by StartPos.

The maximum number of ancestors each input node may have is $h - 1$; the number of combination possible of one node from each *Ilist* is $\prod_{i=1}^{m} n_i$. During the node merging process, for each node (a node from an *IList* or one of its ancestors), we attempt to join it with all the nodes from other *IList*s and their ancestors; for each such merge, one pass is made over the entire set of nodes and ancestors, excluding other nodes from the same *IList* and their ancestors. The time complexity for the merge process thus is $O(h^m \prod_{i=1}^{m} n_i)$. The remaining operations are in proportion to the number of trees generated from the merge process, which is $O(h^m \prod_{i=1}^{m} n_i)$. Hence, the total time complexity of this approach is $O(h^m \prod_{i=1}^{m} n_i)$.

4.2 Efficiently Computing MLCAS

Computing MLCAS using the standard operators, as described above, is simple, but expensive. To efficiently compute MLCASs, we developed a new operator specifically for this purpose, and an evaluation method tailored for it. Our algorithm is inspired by the stack-based family of algorithms for structural join [6, 7, 8, 10], and is limited to XQuery implementations that can support

stack-based structural joins.

Let the position of a node in the XML tree be represented as `(DocID, StartPos, EndPos, Level)`[2], and let each *IList* be sorted by (`DocID`, `StartPos`). The basic idea is to perform one single merge pass over the nodes in *IList*s, in the order of their (start) position in the database tree, and conceptually merge them into rooted trees containing MLCASs. Within each such tree, the root is an MLCA of the inputs, and the leaf level contains all the nodes from different MLCASs sharing the same root. Identification numbers are then used to distinguish nodes from different MLCASs. Each node may have many ancestors: they are not looked up until required. Furthermore, a node is retrieved only once even if it is an ancestor of multiple nodes in the *IList*s.

The main data structure of the algorithm is a stack, with the head of each stack node being a descendant of the head of the stack node below it. Details of the data structure of the stack node are shown in Figure 6. Each stack node is also associated with lists of elements (*Elist*s); each element from *Elist*[i] comes from the corresponding input list *Ilist*[i] ($1 \leq i \leq m$), and has *descendant-or-self* relationship with the head. Some *Elist*s may be marked as *Related* with each other, indicating that the MLCA(s) of nodes from these lists are descendant of the head[3]. Intuitively, one may view a stack node as a tree, with the head being the root, and the elements in the *Elist*s being the leaf nodes.

The full algorithm is shown Figure 7. Here, we walk through it using an example. Consider the XML document in schema A in Figure 1 and Query 3 in Figure 5. For the function **mlcas**($\$y, \$a1, \$t1$), the input lists are *IList*[1]={**3,12**}, *IList*[2]={**6,9,10,15,18**}, and *IList*[3]={**5,8,14,17**}, matching elements *year*, *author*, and *title*, respectively (we ignore term expansion here for the simplicity of illustration). Inputs (nodes) are fetched in ascending order of their `StartPos` and the first input being read is element **3** (a *year*), which is simply pushed onto the empty stack (lines 27-32) (Figure 8(a)).

The algorithm then reads in the next element with smallest `StartPos`, **5** (a *title*), which is not a descendant of the stack top. The current stack top **3** is therefore replaced with its parent **2** (lines 23-26) and added to the *EList*s of **2**. **5** is now a descendant of the new stack top **2**, and is pushed onto the stack (Figure 8(b)). Similarly, when **6** is read in, we replace **5** with its parent **4**, and then push **6** onto the stack (Figure 8(c)). Note that this is a subtle, yet important, optimization to the algorithm: we access an ancestor node only when it is needed to compute MLCASs.

Element **8** is read in next and it is again not a descendant of the stack top **6**. However, at this time, each stack node is a child (not just descendant) of the stack node below it. The stack top and its *Elist*s are therefore recursively appended to the stack node below it (lines 11-22)[4]. Note that a node is retrieved only once even if it is an ancestor of multiple nodes. Such optimization reduces unnecessary index access and contributes to computational saving. With **2** now being the stack top, **8** is pushed onto the stack (Figure 8(d)). Note that the *min* value assigned to **8** is different from that of **5**. The meaning and usage of *min* will be discussed later.

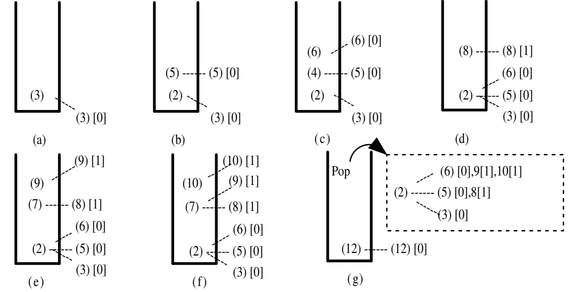


Figure 8: State of stack during evaluation of **mlcas**($\$y, \$a1, \$t1$). Each square bracket contains the *min* value used to distinguish nodes from different MLCASs.

The process of adding **9** and **10** is similar to that of adding **5** and **6** (Figure 8(e) and (f)). When **12** is read in, as what happens with **8**, the stack top and the associated *Elist*s are recursively appended to the node below it. Finally, stack top **2** is found to contain no empty *Elist*s (indicating that it contains MLCASs), and popped as output. It is guaranteed that all the MLCASs sharing **2** as the root have been found (in the *Elist*s). We then push **12** onto the empty stack (Figure 8(g)). The algorithm continues until there is no input element and the stack is empty.

Identification numbers $[min, max]$ are used to distinguish different MLCASs. *min* is assigned for each input element when it is added to the stack (lines 34-45), while *max* equals $min(nextMin-1, \infty)$, where $nextMin$ refers to the *min* value of the next element in the same list. Elements from *Related Elist*s with compatible identification numbers, i.e., the intersection of their identification numbers is non-empty, belong to the same MLCAS(s), while element from not *Related Elist*s may belong to the same MLCAS(s), regardless of their identification numbers. When a node is popped from the stack with associated *EList*s, such numbers are used to identify nodes (in *Elist*s) belonging to the same MLCAS and construct MLCASs.

Theorem 4.2 *The time complexity of the stack-based MLCAS algorithm is* $O(h\sum_{i=1}^{m} n_i + \prod_{i=1}^{m} n_i)$, *where* h *denotes the height of the XML data tree.*

The intuition is as follows. Each input element, and its ancestors, may be pushed onto the stack at most once, and when on the stack, be popped from stack, appended to, or deleted from an *Elist* associated with another node at most once (the *EList*s are implemented as linked lists, with start and end pointers; appending or deletion can

[2]`DocID`: the identifier of the document; `StartPos`/`EndPos`: generated by counting word numbers from the beginning of the document until the start of the element and the end of the element, respectively; `Level`: the nesting depth of the element. Notice that a node can be identified by the pair (`DocID`, `StartPos`).

[3]Bitset array relBits is used to denote which lists are related with each other at a level lower than that of the head node. Due to space limitation, we use the simple notion of marking lists as *Related*, and will not discuss the details of how to manipulate relBits in this paper.

[4]First add **6** and its *Elist*s to **4**; then add **4** and its *Elist*s to **2**; finally, **4** is removed since it does not belong to any *IList*.

be performed in unit time). Since each stack operation falls into the one of those constant time operations, the time complexity is $O(h\sum_{i=1}^{m} n_i)$. Finally, the time required for merging MLCASs from the output trees is linear in the output size. In the worst case, all MLCASs share the same root and each node in a list is meaningfully related with every node from other lists. In such a case, the time required for the merge process will be $O(\prod_{i=1}^{m} n_i)$. Putting all together, we get a time complexity of $O(h\sum_{i=1}^{m} n_i + \prod_{i=1}^{m} n_i)$ for our stack-based algorithm. Clearly, no competing algorithm that has the same input lists, and is required to compute the same output, could have better asymptotic complexity, since each input has to be read and each output has to be computed.

5 Experimental Evaluation

We implemented Schema-Free XQuery using the Timber native XML database [1, 19] and evaluated the system on two aspects: 1) search quality, which is evaluated using both a standard XML benchmark (Section 5.1) and a heterogeneous data collection (Section 5.2); 2) search performance, where we measure the overhead caused by evaluating schema-free query versus the schema-aware query (Section 5.3).

Throughout this section, the quality of a search technique was measured in terms of accuracy and completeness using standard precision and recall metrics, where the correct results are the answers returned by the corresponding schema-aware XQuery. Precision measures accuracy, indicating the fraction of results in the approximate answer that are correct, while recall measures completeness, indicating the fraction of all correct results actually captured in the approximate answer.

We note here that information retrieval systems can usually trade off precision against recall by choosing a different threshold value for a scoring function used to evaluate candidate results. A high threshold will return results only with a high score, giving good precision at the expense of recall. A low threshold will have the opposite effect. Evaluation of IR systems usually includes a precision-recall curve representing this tradeoff. Schema-free XQuery is still a database query language, and does not use any scoring functions in its evaluation. As such, there is no possibility of returning more or fewer results, and so no possibility of establishing a precision-recall curve.

5.1 Search Quality: XMark

XMark: XMark is a popular benchmark and its queries pose a wide range of challenges: from stressing the textual content of the document to ad-hoc data analysis [3]. We generated the XMark data set using a factor of 0.45, which had 1.45 millions of nodes and occupied 179 MB when loaded into our database. Indices with a total size of 106MB were also built.

To evaluate the relative strength of Schema-Free XQuery, we compared it with two techniques that support search over XML documents without knowledge of XML schema: Meet [22] and XSEarch [11]. Meet proposes to find the LCA for the set of keywords given in the query and return the subtree rooted at the LCA as the answer to the query. XSEarch is considered superior to a pure keyword based approach as it distinguishes tag names from textual content and has a better way of determining meaningful relationships among nodes based on the document structure (for our comparison, we adopted the all-pairs strategy of XSEarch, which is more competitive in search quality).

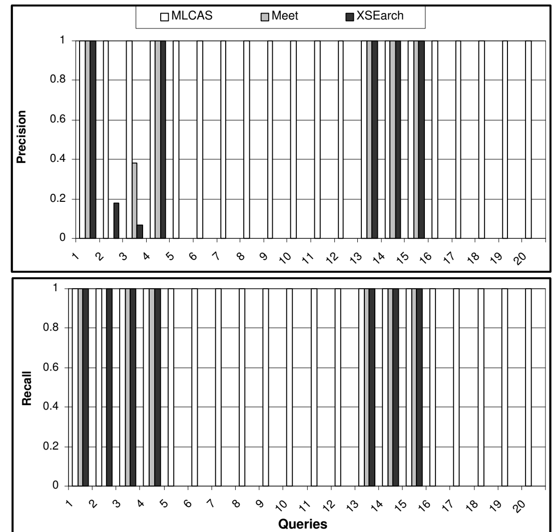


Figure 9: Precision and recall of different search strategies on XMark. Missing bars indicate a value of zero.

We expanded each original natural language query into a keyword search query, an XSEarch query, and a Schema-Free XQuery. We also wrote a schema-aware XQuery for each query and each XML document (different documents have different schema and a schema-aware XQuery has to be constructed for each of them). We obtained the correct answers by running the schema-aware XQuery and additionally verified correctness manually.

Result: Figure 9 presents the precision and recall of the three techniques for XMark. Schema-free XQuery (MLCAS) achieved perfect precision and recall for all the queries (i.e., all the results returned by **mlcas**-embedded XQuery were correct and all the possible correct results were returned). In contrast, Meet and XSEarch performed poorly on many of the queries, especially those with dynamic search conditions, or requiring complex manipulations such as ordering or grouping (Queries 5, 6, etc.). In particular, the root of the structure returned by Meet is on average 3 levels higher than the root of the correct structure: this observation indicates that a simple subtree rooted at LCA of the keywords, although usually covers the correct segments of interest, too often includes much irrelevant information, and cannot be easily manipulated to generate correct answers. Even for queries with simple constant search conditions and requiring no further manipulation (Queries 1, 4, etc.), Meet and XSEarch often produce results that are correct but too inclusive (we have counted those as correct answers in the Figure 9): unrelated elements are returned along

with the meaningful ones.

5.2 Search Quality: Publication Collection

In working with XMark, we certainly knew its schema. We tried not to let this influence our specification of Schema-Free XQuery, and believe that we were successful in this. Nevertheless, a skeptic may have reason to be suspicious of our results. One way to address this concern is to work with heterogenous schema. But now we face the problem that there is no standard heterogeneous XML benchmark, so we decided to focus on a set of meaningful queries and search for a collection of heterogeneous data set to accommodate them. Queries from XMark were considered first, but unfortunately, real-world auction data required by XMark were not publicly available. We noticed, however, that "XMP," a comprehensive set of queries from XQuery use case [24], were largely based on bibliography documents, which were relatively easy to collect from the web. We therefore decided to use the 11 queries[5] from "XMP," plus an example query (also based on bibliography data) from XSEarch [11] for this part of evaluation.

Publication collection: We manually collected personal publication lists from 300 faculty personal homepages in a large research university[6] to serve as the data set for the "XMP" queries. Those publication lists, while obtained from the real world, are semantically close enough to the bibliography data such that our "XMP" query set can be applied with only minor changes (e.g., tag name *year* is used to replace *price*, which is not in the data set but has similar characteristics). These publication lists, despite the similarity in their semantics, vary greatly in terms of structure and normalization rules. In fact, if we rewrite them into XML documents, a total of 72 distinct schemas are found. However, many of these schemas either have equivalent structures or only differ from each other in minor details (e.g., a few include abstracts while most do not). If we group the lists based on their structural similarity, the union of the schemas of the lists within each group can then be used to represent all the lists in the same group. We refer to each group as a *schema family*. Schemas within a schema family are similar and therefore tend to have similar effects on the search quality for different search techniques. We identified six schema families for the 300 personal publication lists collected, and present results for one representative document from each.

Result: Figure 10 shows the average[7] precision and recall of the three techniques over the set of "XMP" queries against the publication collection. For all the queries, Schema-Free XQuery achieved perfect precision and recall, while Meet and XSEarch had poor precision and recall for many queries. This result demonstrates the robustness of Schema-Free XQuery against changes in document schema, considering that for each original

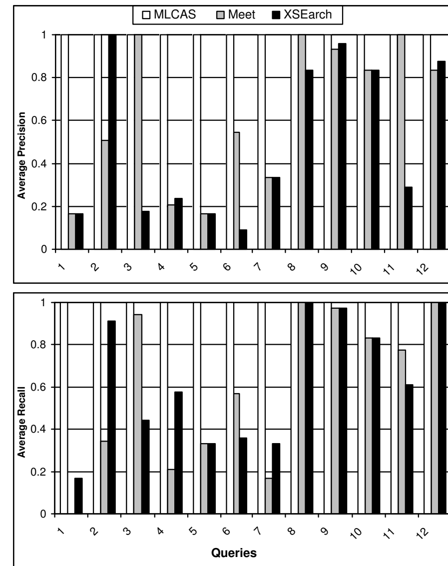


Figure 10: Average precision/recall of different search strategies for publication collection with term expansion. Missing bars indicate a value of zero.

natural language query, we ran exactly the same schema-free query on all the publication lists.

Although Schema-Free XQuery achieved 100% precision and recall for all of our queries, it does not imply that Schema-Free XQuery guarantees such perfect search quality for any dataset and/or any query. For instance, if we change the XML document shown in Figure 1(A) such that *author* node **6** and *title* node **8** are removed, for Query 1 in Figure 1, Schema-Free XQuery will return (**5**,**3**) as the result, while the correct answer should be (`empty`,**3**). Our extensive experimental evaluation suggests that such instances are uncommon.

Term expansion was employed for all the three strategies investigated in this comparison. The absence of term expansion reduced the average precision and recall of about half of the queries for all three strategies (Figure 11). It is not a surprise to see that a mismatch on even one single tag name could reduce the search quality significantly. If no nodes with the correct tag name can be found, one can obviously not find MLCAS, all-pair R answer, or LCA.

5.3 Search Performance

We measure the performance of Schema-Free XQuery in terms of simplicity and efficiency. To evaluate simplicity, we compare the number of operators in the evaluation plan generated for the **mlcas**-embedded XQuery and the corresponding XQuery, with **mlcas** computation being considered a single operator. To evaluate efficiency, we compare the time cost of evaluating an **mlcas**-embedded XQuery, with both the basic and the stack-based implementation of MLCAS computation, with that of evaluating a schema-aware, fully specified XQuery. For these experiments, the XMark data set worked fine, but the heterogeneous publication collection was too small to be interesting. Instead, we used the DBLP data set [20], which was of sufficient size to show non-trivial running

[5] Q12 is not included since set comparison is not yet supported in Timber.

[6] This includes all the personal homepages from four departments (123 in all), and a few randomly chosen personal homepages from 21 other departments (177 in all).

[7] Over the six representative documents

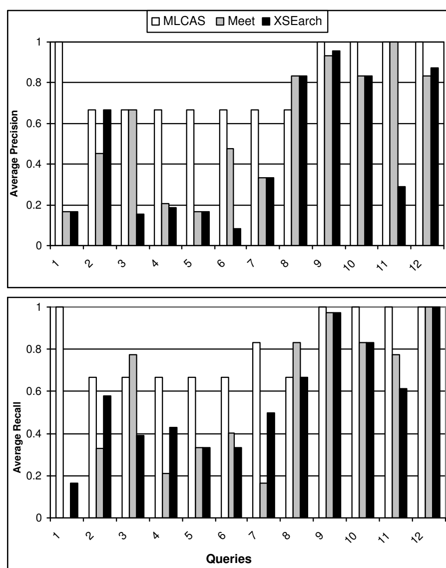


Figure 11: Average precision/recall of different search techniques for publication collection without term expansion.

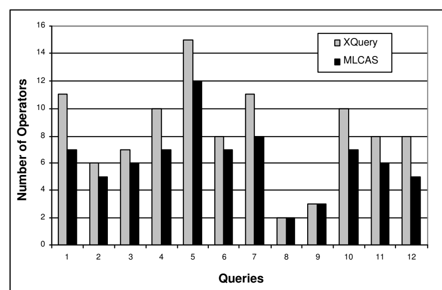


Figure 12: Average number of operators in evaluation plans generated by schema-aware XQuery (XQuery) and **mlcas**-embedded XQuery (MLCAS) for "XMP" queries on DBLP.

Query	*XQuery*	*MLCAS*	*COMPOSE*
1	530	1533	25621
2	290	1173	DNF
3	527	1421	DNF
4	479	1665	26591
5	1132	2518	DNF
6	371	1116	DNF
7	552	1469	24590
8	240	243	241
9	237	240	240
10	367	1456	24536
11	511	1321	DNF
12	473	1088	DNF

Table 1: Performance (seconds) of XQuery, MLCAS, and COMPOSE for the "XMP" queries on DBLP.

time while still within the bibliography domain such that the queries evaluated in the experiments above could apply. This data set comprised nearly 86 millions nodes, and occupied 957 MB for the data and 437 MB for the indices when loaded into our database.

The experiments were carried out on a Pentium III PC machine (800 MHz CPU, 512MB RAM, 120GB hard drive) running Windows 2000 Professional. The Timber buffer size was set to 64KB. We excluded the time for query parsing and evaluation plan generation in all the cases. Each query was run five times for each XML document with a cold operating system cache. The average running time was used in the performance evaluation. Note that for COMPOSE (the basic implementation for computing MLCAS previously discussed in Section 4.1), the execution time for some queries is marked as DNF, which means that the execution was killed when it did not finish within 7 hours.

Results: Figure 12 shows that evaluation plans generated by **mlcas**-embedded XQuery for the "XMP" queries on DBLP data are usually simpler than those of XQuery, with 2 fewer operators on average, an approximately 25% savings in plan generation. Furthermore, unlike the schema-aware XQuery, the same evaluation plan can be generated once and used over multiple documents with different schemas.

Table 1 reports the actual execution time of **mlcas**-embedded XQuery, both the stack-based algorithm (MLCAS) and the basic algorithm (COMPOSE), and schema-aware XQuery (XQuery) for the "XMP" queries on DBLP data. Our stack-based MLCAS algorithm speeds up the processing of **mlcas**-embedded XQuery by approximately 16 times, often reducing the execution time from more than 7 hours to less than 30 minutes. The capability of Schema-Free XQuery does not require expensive cost in performance. The overhead of **mlcas**-embedded XQuery using MLCAS algorithm is between 100% and 300%, with the exception of Q8 and Q9. There is no overhead for these two because they involve only one tag name, and thus no computation of MLCASs is needed. The existence of such overhead is expected. **mlcas**-embedded XQuery usually has to process more data than its schema-aware counterpart: the filtering of results according to the search conditions is done after the computation of MLCASs, while in schema-aware XQuery, most such filtering is done at data fetching time. In our future work, we will exploit optimization techniques to reduce such overhead.

Results for XMark are similar: over 20 different query types, the geometric mean of the running time of **mlcas**-embedded XQuery is 26.2 seconds, while that of schema-aware XQuery is 12.3 seconds. We cannot compute the geometric mean of the running time for COMPOSE, as 11 out of 20 queries failed to finish within 7 hours. The overhead for **mlcas**-embedded XQuery, compared to schema-aware XQuery varied from 0% to 250%.

6 Related Work

Extensive research has been done on structured declarative queries as well as on keyword based text search. In recent years, there has been interests in techniques that merge the two. BANKS [4], DBXplorer [5], and DISCOVER [18] attempt to apply keyword search on relational database. In those studies, a database is viewed as a graph with objects/tuples as nodes and relationship as edges, and sub-graphs of the database are returned as answers to the original keyword query. Similar approach has also been taken to apply keyword search in XML documents (e.g., XKeyword [17] and XRANK [15]). Ranking mechanisms have been applied to the search results such that results with perceived higher relevance are returned to the user first. All such keyword search approaches suffer from two drawbacks: (1) they do not distinguish tag name from textual content; (2) they can-

not express complex query semantics.

A number of attempts have also been made to support information retrieval style search by expanding XQuery [9] or other structured query languages (e.g., XXL [23], XIRQL [13], and [12]). These approaches require a user to learn the query semantics and in cases where a user is unaware of the document structure, they do not exploit any document structure. Other approaches (e.g., LOREL [21] and Meet [22]) created query languages to enable keyword search in XML documents and exploit some structural information that is not specified in the query. The differences between those approaches and ours are that we eliminate any requirement for path expressions, and we exploit the document structure better to identify results that are more meaningful.

A recent closely related work is XSEarch [11], which attempts to return meaningful results based on query as well as document structure using a heuristic called *interconnection* relationship. In XSEarch, two nodes are considered to be semantically related if and only if there are no two distinct nodes with the same tag name on the path between these two nodes (excluding the two nodes themselves). Queries are allowed to specify tag names and attribute value pairs. However, *interconnection* does not work when two unrelated entities are present in entities of different types. For example, two *author* nodes may be considered as interconnected, even though one of them belongs to an *article* node and the other belongs to a *book* node. Moreover, due to the simple query semantics used, XSEarch suffers from drawbacks similar to keyword search methods: difficulty to express complex knowledge semantics. The MLCAS operator, on the other hand, takes full advantage of well-defined XQuery, and enables the user to take more control of the search results without knowing the document structure.

Finally, the REVERE system allows query answering across schemas by deploying schema mapping and query rewriting techniques [16]. Users are still required to have extensive knowledge of at least one schema to pose queries. No experimental evaluation on the effectiveness of the system has been reported.

7 Conclusion

The main contribution of this paper is to show that a simple, novel XML document search technique, namely Schema-Free XQuery, can enable users to take full advantage of XQuery in querying XML data precisely and efficiently without requiring full knowledge of the document schema. At the same time, any partial knowledge available to the user can be exploited to advantage. We have shown that it is possible to express a wide variety of queries in a schema-free manner and have them return correct results over a broad diversity of schema. Given its robustness against schema changes, Schema-Free XQuery is potentially of value in a data integration or data evolution context where one would like a query written once to apply "universally" and "forever". We also devised a stack-based algorithm for the MLCAS computation at the heart of schema-free query. Experiments showed that this algorithm was up to 16 times faster than a basic MLCAS computation using standard operators. Schema-free queries evaluated with this stack-based algorithm incurred an overhead no more than 3 times the execution time of an equivalent schema-aware query. Future directions for research include ontology-driven term expansion and further optimization of query processing by exploiting possibilities of pushing MLCAS calculation further down in the evaluation plan. We also intend to investigate techniques for applying MLCAS to queries involving attributes and references. Finally, we intend to use more sophisticated IR techniques where appropriate in schema-free queries.

References

[1] TIMBER: http://www.eecs.umich.edu/db/timber.

[2] WordNet: http://www.cogsci.princeton.edu/~wn/.

[3] XMark: http://monetdb.cwi.nl/xml/index.html.

[4] B. Aditya et al. BANKS: Browsing and keyword searching in relational databases. In *VLDB*, 2002.

[5] S. Agrawal et al. DBXplorer: a system for keyword-based search over relational databases. In *ICDE*, 2002.

[6] S. Al-Khalifa et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2001.

[7] S. Al-Khalifa et al. Querying structured text in an XML database. In *SIGMOD*, 2003.

[8] N. Bruno et al. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[9] D. Chamberlin. XQuery: An XML query language. *IBM System Journal*, 41:597–615, 2003.

[10] S.-Y. Chien et al. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[11] S. Cohen et al. XSEarch: A semantic search engine for XML. In *VLDB*, 2003.

[12] D. Florescu et al. Integrating keyword search into XML query processing. *Computer Networks*, 33:119–135, 2000.

[13] N. Fuhr and K. Großjohann. XIRQL: An extension of XQL for information retrieval. In *SIGIR*, 2000.

[14] G. W. Furnas et al. The vocabulary problem in human-system communication. *CACM*, 30(11):964–971, 1987.

[15] L. Guo et al. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[16] A. Halevy et al. Crossing the structure chasm, 2003.

[17] V. Hristidis et al. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[18] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[19] H. V. Jagadish et al. Timber: A native xml database. *The VLDB Journal*, 11(4):274–291, 2002.

[20] M. Ley. DBLP bibliography, 2003.

[21] D. Quass et al. Querying semistructured heterogeneous information. In *DOOD*, 1995.

[22] A. Schmidt et al. Querying XML documents made easy: Nearest concept queries. In *ICDE*, 2001.

[23] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT*, 2002.

[24] W3C. XML query use cases, 2003.

[25] W3C. XML schema, 2003.