The VLDB Journal (2006)
DOI 10.1007/s00778-006-0003-4

REGULAR PAPER

Yunyao Li · Cong Yu · H. V. Jagadish

Enabling Schema-Free XQuery with meaningful query focus

Received: 29 November 2004 / Accepted: 8 August 2005

Abstract The widespread adoption of XML holds the promise that document structure can be exploited to specify precise database queries. However, users may have only a limited knowledge of the XML structure, and may be unable to produce a correct XQuery expression, 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 Query Focus (MQF) for finding related nodes within an XML document. MQF enables users to take full advantage of the preciseness and efficiency of XQuery 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 data integration or data evolution. 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 is possible to express a wide variety of queries in a schema-free manner and efficiently retrieve 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 developed for computing MQF, the overhead is from 1 to 4 times the execution time of an equivalent schema-aware query. The evaluation cost of schema-free queries can be further reduced by as much as 68% using a selectivity-based algorithm we develop to enable the integration of MQF operation into the query pipeline.

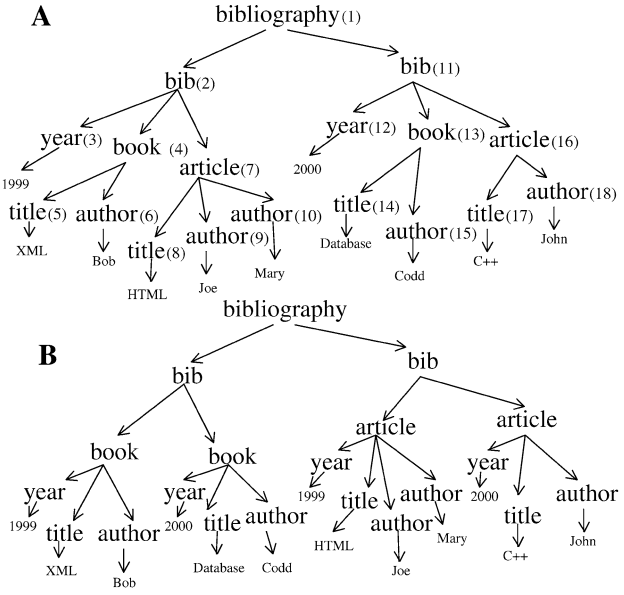
Keywords Hierachical · Semi-structured · XML · Schema · Query language · XQuery

Y. Li (✉) · C. Yu · H. V. Jagadish
Department of EECS, University of Michigan, Ann Arbor, MI 48109, USA
E-mail: {yunyaol, congy, jag}@umich.edu

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 two paths: a structured query approach or a keyword-based approach. XQuery [14] is the generally acknowledged standard of the former, while the latter class has several recent suggestions, including XKeyword [25] and XSEarch [17]. Both approaches have their advantages and disadvantages. Fully structured query (e.g., XQuery), working with the database schema, can convey complex semantic meaning in the query, and therefore can precisely retrieve desired results. However, if the user does not know the (full) database structure, it is difficult to write a correct query. Even if the user does know the schema, 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 Fig. 1 showing the same bibliography data arranged in two different schemas: A organizes publications based on the publication *year*, while 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*



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.

Fig. 1 Querying XML data with multiple schemas

and *author* are the child elements of "publication", while *year* could be either a 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 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 the desired user intention. Of course, it is straightforward to express each of these three example queries using XQuery – but doing so requires knowledge of the database schema, which the user may not possess.

In this paper, we develop a framework that enables users to query XML data, exploiting whatever partial knowledge of the schema they have. If they knows the full schema, they can write regular XQuery statements. 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 [34]) as an effective mechanism to identify segments of the database of interest to a pure keyword query. While this intuition is reasonable, we show in Sect. 2 that LCA can frequently be too inclusive. We refine LCA and define the concept of a *Meaningful Query Focus* (MQF), which is an XML fragment that meaningfully relates nodes corresponding to relevant variables in the XQuery expression. We also illustrate through examples how this structure, and its root node, can be referenced and manipulated in an XQuery expression with embedded mqf functions.

In Sect. 3, we propose Schema-Free XQuery as a suitable query language for flexible querying of heterogeneous information. It incorporates MQF into XQuery with a straightforward query logic and enables users with limited schema knowledge to write simplistic XQuery expressions. In fact, users can write a query specifying keywords, 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.

MQF computation is the core part of Schema-Free XQuery evaluation. In Sect. 4, we show how to accomplish this using standard XQuery evaluation operators. We then introduce a novel stack-based algorithm to compute MQF more efficiently, in a manner reminiscent of containment join. In Sect. 5, we present the 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 what an equivalent schema-aware query would require.

Schema-Free XQuery, as described up to this point, is a stand-alone operator. In Sect. 6, we investigate the possibility of integrating MQF computation into a query evaluation pipeline. We present a modified notion of MQF to fit into this context. In addition, we propose an *Ancestor–Descendant Summarization* (A–D) Index to allow efficient access to base data information. In Sect. 7, we develop two different algorithms, both extensions to the original stack-based algorithm (Sect. 4), to provide further acceleration for the evaluation of MQF. Our comparison study in Sect. 8 shows that the integration of MQF into a query pipeline supported by our new algorithms can significantly reduce the evaluation cost of Schema-Free XQuery for a wide arrange of queries, especially for queries with selective selection conditions.

Finally, we discuss related work in Sect. 9 and conclude in Sect. 10.

2 Meaningful query focus

In this section, we describe the concept of MQF and show the use of the mqf function in an XQuery expression. We begin with a few preliminaries.

We model an XML document as a rooted, ordered, and labelled tree. Nodes in this rooted tree correspond to elements in the XML document.

Definition 1 (Descendant-Or-Self) A tree node n_d is said to have a descendant-or-self relationship with another tree node n_a, if it is a descendant of n_a or is equal to n_a, denoted as $n_d \preceq n_a$.

Definition 2 (CA) Let the set of nodes in an XML document be N. For $d_1, d_2 \in N$, $a \in N$ is a Common Ancestor (CA) of d_1 and d_2, denoted as CA(d_1, d_2), if and only if $d_1 \preceq a$, and $d_2 \preceq a$.

Definition 3 (LCA) Let the set of nodes in an XML document be N. For $d_1, d_2 \in N$, $a \in N$ is the Least Common Ancestor (LCA) of d_1 and d_2, denoted as LCA(d_1, d_2), if and only if:

- $a = \text{CA}(d_1, d_2)$, and
- $\forall\, a' \in N\ (a' \neq a)$, if $a' = \text{CA}(d_1, d_2)$, then $a \prec a'$.

2.1 Motivation for MQF

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 Fig. 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 [17, 25] and works well in certain cases. For example, consider nodes **8** (*title*) and **10** (*author*) in Fig. 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 Fig. 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 MQF, and using it as the refined context for query evaluation.

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

Fig. 2 Query 1 in XQuery with no structural knowledge

2.2 MLCA

A node in an XML document, along with its entire subtree, typically represents a real-world entity. The tag name identifies the type of the entity, which is called *entity type* to distinguish it from the data type used by XML Schema [37]. In the presence of domain ontology, nodes with different tag names may still be regarded as of the same entity type. For example, *book* and *article* nodes may be deemed as of the same super-type *publication*. Such ontology-based tag name matching is presented in detail in Sect. 3.2 and is not considered here to allow for a clear presentation of the MQF concept.

We now describe the intuitive notion of two nodes being meaningfully related to each other. Consider the diagrams in Fig. 3, let n_1 and n_2 be two nodes in the XML document with different entity types. There are two possible structural relationships between them. First, n_1 is an ancestor node of n_2 (Fig. 3a), or vice versa. We believe, in this case, that n_1 and n_2 are meaningfully related to each other. Second, n_1 and n_2 have no hierarchical relationship with each other. Suppose the LCA of n_1 and n_2 is node n (Fig. 3b), 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 Fig. 3c, 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'.

Consider the previously mentioned example of nodes **5** and **10** in Fig. 1, with node **2** being their LCA. However, node **2** 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 MQF) because they belong to different publications.

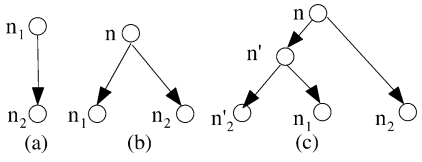


Fig. 3 Structural relationships among nodes

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 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 $c_k = \text{LCA}(a_i,b_j)$; c_k is denoted as $\text{MLCA}(a_i, b_j)$.
- $\forall\, a_i \in A, b_j \in B$, for $d_{ij} = \text{LCA}(a_i, b_j)$, if $d_{ij} \notin C$, then $\exists\, c_k \in C, c_k \prec d_{ij}$, and vice versa.

The set C is denoted as $\text{MLCASET}(A, B)$.

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 b, belongs to C, where C is $\text{MLCASET}(A, B)$. 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 relates the entities represented by nodes in A and B more closely. Given multiple sets of nodes, where nodes within each set are of the same type, we can easily extend Definition 4 to define the MLCA of multiple nodes:

Definition 5 (MLCA of multiple nodes) Let the set of nodes in an XML document be N. Given $A_1, A_2, \ldots, A_m \subseteq N$, where A_i is comprised of nodes of type $\mathcal{A}_i$ ($i \in [1, \ldots, m]$), a Meaningful Lowest Common Ancestor $c = \text{MLCA}(a_1, \ldots, a_m)$ ($a_i \in A_i$, $i \in [1, \ldots, m]$) satisfies the following conditions:

- $\forall i,j \in [1, \ldots, m]$ $(i \neq j)$, $\exists\, m = \text{MLCA}(a_i, a_j)$, where $m \neq \texttt{null}$ and $m \preceq c$.
- $\exists i,j \in [1, \ldots, m]$ $(i \neq j)$, $c = \text{MLCA}(a_i, a_j)$.

Consider the simple XML document in Fig. 4a. Suppose we want to find MLCAs for all nodes of type *title*, *author*, *review*. The only MLCA found is node **1**, MLCA(**2**,**3**,**4**). It satisfies both conditions in Definition 5, as MLCA(**2**,**3**) = MLCA(**3**,**4**) = MLCA(**1**,**4**) = **1**. If we remove the first *author* node (**3**) from the XML document (for the ease of comparison, ids of the rest nodes remain the same), we can obtain a new document as shown in Fig. 4b. However, in this new document, we can no longer find any MLCAs for nodes of type *title*, *author*, *review*. Node **0** is the only node with descendants of all the three types. Yet it is not an MLCA that we are looking for, as it does not satisfying Definition 5 for any combination of nodes of the three types. For example, **0** is not MLCA(**2**,**7**,**4**), because MLCA(**2**,**7**) does not exist, violating the first condition of Definition 5 (node **0** is not MLCA(**2**,**7**), as MLCA(**6**,**7**) = **5** $\prec$ **0**). Similarly, it is not MLCA(**2**,**8**,**4**), MLCA(**6**,**7**,**4**) or MLCA(**6**,**8**,**4**).

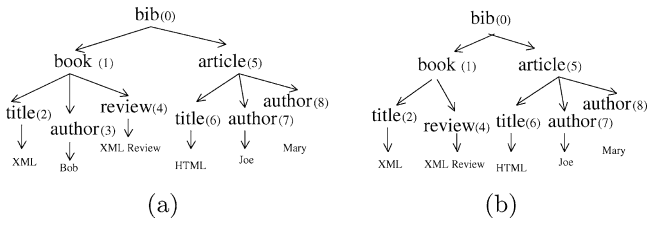


Fig. 4 Example XML data: compute MLCA of multiple nodes

2.3 MQF

Finding MLCA for multiple nodes, however, 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 Fig. 2 against the data in schema A in Fig. 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*, so 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 the entire structure, MQF, for each such established relationship.

Definition 6 (MQF) Let the set of nodes in an XML document be N. Given $A_1, A_2, \ldots, A_m \subseteq N$, where A_j is comprised of nodes of type $\mathcal{A}_j$ ($j \in [1, \ldots, m]$), the Meaningful Query Focus Set $S = \{(r, a_1, \ldots, a_m) \mid r \in N, a_i \in A_i\ (i \in [1, \ldots, m]), r = \text{MLCA}(a_1, \ldots, a_m)\}$. Each element of this set is denoted as $\text{MQF}(a_1, \ldots, a_m)$, with r as its root.

Each MQF is used as a refined context for query evaluation, since it contains only the nodes that are meaningfully related to one another. An MQF is the tightest structure containing one node from each input set. If an MQF satisfies the search conditions, it is unlikely to contain a wrong answer. For example, for Query 1 in Fig. 1, expressed as shown in Fig. 2, we obtain several MQFs, including (**2**,**10**,**8**,**3**) and (**11**,**15**,**14**,**12**). The only MQF 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 MQF and the concept of interconnected nodes employed by the XSEarch system [17]. 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

```
Query 1:
  for  $a in doc("bib.xml")//author,
       $t in doc("bib.xml")//title,
       $y in doc("bib.xml")//year
  where  $a/text() = "Mary" and mqf($a,$t,$y)
  return  <result> {$t, $y} </result>

Query 2:
  for  $a in doc("bib.xml")//author,
       $b in doc("bib.xml")//author
  where  $a/text() = "Mary" and $a != $b
         and mqf($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  $a/text() = "Mary" and mqf($a,$t)
         return  $t }
  let  $m := mqf($y,$a1,$t1)
  where  $t1 ≈ $t2 and exists $m
  return  <result> {$y, $a1} </result>
```

Fig. 5 Example queries with mqf function

nodes are a 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.

2.4 Adding mqf function to XQuery

In this section, we introduce a new function, mqf, to the standard XQuery language:

Definition 7 (mqf **function)** mqf($a_1, \ldots, a_n$) is a function that returns (i) the root node of MQF($a_1, \ldots, a_n$), if it exists, (ii) `null` otherwise.

Figure 5 shows how each of the three running queries presented in Fig. 1 can be expressed in the XQuery enriched with the mqf 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 mqf($\$a, \$t, \$y$) defines the context for evaluation to be the MQF of those nodes and filters out any node that cannot be part of any MQF. The query is flexible since it does not require the user to know the exact relationships between nodes of the three types.

Query 2 shows another aspect of the flexibility in the mqf function: the individual nodes do not have to be of different types. By combining the conditions $\$a \; ! = \$b$ and mqf($\$a, \$b$), the only MQFs 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[1] $\$t1 \approx \$t2$. 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 in the runtime. Although the binding of the result of mqf function to a variable $\$m$ is not necessary, it is shown here to illustrate that the root of the MQF can be manipulated just like any other regular elements in the XML document. If we evaluate this query against data in schema A of Fig. 1, the only MQF 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 MQFs, including (**2**,**3**,**6**,**5**), (**2**,**3**,**9**,**8**), (**2**,**3**,**10**,**8**), (**11**,**12**,**15**,**14**), and (**11**,**12**,**18**,**17**). Only the second and third MQF have *title* similar to "HTML" and are the final MQFs when we consider the similarity join. The final results to be returned are therefore (*year* = "1999", *author* = "Joe"), and (*year* = "1999", *author* = "Mary").

3 Schema-Free XQuery

While the use of mqf function inside XQuery allows the user to issue a query without knowing the exact structure of the document, understanding the semantics of the function and effectively using it can become quite a burden on the user. In Sect. 3.1, we propose a simpler query logic and the use of a related keyword to reduce the complexities involved in writing such an mqf-embedded XQuery. Another issue in practice is *term 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 briefly address this issue in Sect. 3.2. The final result is a Schema-Free XQuery that a user can write easily and without knowing the exact schema.

3.1 MQF transformation

The use of mqf function in XQuery allows the user to group semantically related entities without knowing how they are related in the document. However, the semantics of such a function may be difficult for naive users, our primary target, to understand and use correctly. We observe that in most queries, a single query block (i.e., a FLWOR block, excluding any other query blocks nested in it) typically contains one meaningful query focus. We call these *simple focus*

[1] The similarity join is assumed to be a user provided boolean function that decides whether two strings are similar or not.

queries.[2] For such queries, we propose to further reduce the complexity of mqf-embedded XQuery by replacing the mqf function with a simple related keyword. Instead of writing out the mqf function explicitly, users simply identify the set of related entities in a single query block using the keyword related. Our system will then automatically transform the simplistic XQuery with related keyword into one with embedded mqf function. The following query illustrates a simple XQuery with related keyword representing Query 1 in Fig. 1:

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

This query can be automatically transformed into Query 1 in Fig. 5 by creating an mqf function in the where clause of the query, with all the related marked variables being its arguments. Another example shows how MQFs in nested queries can be identified:

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

As previously mentioned, MQF is designed to have a scope that is local to a single query block. Hence, the MQF formed from the related marked variables in the inner query is different from the MQF formed from the related marked variables in the parent query. The two MQFs are linked together by the similarity join and our system will transform this query into Query 3 in Fig. 5. Furthermore, the related marked variables do not necessarily 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 related $r//author,
     $b in related $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 putting the two related marked *author*s under $r. The following transformed query will be produced:

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

When evaluating the transformed query, the system will take all *author*s that are descendants of the first *bib* element and automatically compute MQFs from those nodes only.

Remarks XQuery extended with the related keyword is designed to serve a naive user who has little knowledge of the database or database query language. With this keyword, one only needs to think of one concept group at a time and use the keyword to flag the groups out. On the other hand, the mqf-embedded XQuery is more flexible and allows multiple concept groups on the same level at the cost of exposing greater complexity to the user.

XQuery Extension: We propose the following extension to the standard XQuery language [38], which adds the related keyword into the expression of ForClause.

```
ForClause ::= "for" "$" VarName TypeDeclaration?
    PositionalVar? "in" ( ("related" PathExpr) |
    (ExprSingle (", " "$" VarName TypeDeclaration?
    PositionalVar? "in" ExprSingle)*) )
```

This extension allows the related keyword to be placed before any PathExpr in the for clause of a (sub-)XQuery to indicate that the preceding variable will bind to a data node that must be considered part of the meaningful query focus.

Transformation Algorithm: The transformation algorithm works in a straightforward way. It identifies all variables marked with related keyword within a single query block, and creates an mqf function in the where clause of that block with those variables as the arguments. Each nested query block with at least one related marked variable will have its own mqf function.

3.2 Expanding terms

While the flexibility of MQF helps to address the issue of structure ambiguity, users still have to rely on the correctness of tag names (called *terms*) used in a query to produce desired results. For example, if the document being queried uses *au* instead of *author* to denote the concept of author, none of the running example queries will generate the correct results. In an *ad hoc* information retrieval task, a casual user is as unlikely to have perfect knowledge of tag names as to have perfect knowledge of structure relationships. We call this issue *term ambiguity*: the discrepancy between a query term and its actual tag name counterpart in the document. There are two main reasons for this discrepancy: first, the common use of interchangeable synonyms to describe the same concept; second, the domain specific usage of hypernyms (terms that are more generic than the given term) and hyponyms (terms that are more specific than the given term) that are different in scope with the terms in the database.

[2] All example queries in Fig. 5 fall into this category.

To address this term ambiguity issue, we expand user provided tag names using two strategies. The first is synonym-based expansion. Given a thesaurus of synonyms, we group all synonyms together and choose one as the standard term. Each element tag name is indexed both as itself and as the standard term. At query time, terms in the query can then be normalized into the standard term and matched with itself or any synonym it might have. The second is ontology-based expansion. Given a domain ontology, the hyponyms and hypernyms (if not already element tag names themselves) of each element tag name are also indexed for matching at the query time. With the adoption of expanded index, instead of being turned into multiple queries in order to handle synonyms, hyponyms, and hypernyms, each user query is simply converted into its standard form and evaluating the standard query alone will be sufficient.

In practice, one naive approach to term expansion is to expand all the terms encountered in the user's query. Such an approach, we believe, is in violation of the principle of Schema-Free XQuery, which is to help the user construct meaningful queries when the knowledge of schema (in terms of both structural and term ambiguity) is missing, while giving the user the power to express the exact meaning when the knowledge of schema is present. Unnecessary expansion of user terms (i.e., expanding terms when the users specify the exact term they want) can potentially overwhelm the users with many irrelevant results. As a result, we propose the addition of a simple function: expand. The expand function, when applied to an element tag name specification, indicates that the term is to be expanded during the query evaluation process. Consider the following query:

```
for  $b in doc("bib.xml")//expand(publication),
     $y in doc("bib.xml")//year
where  $y = "1999" and mqf($b, $y)
return  $b
```

The tag name *publication* in the query is indicated by the expand function as being not exact, and can be matched to its synonyms in the thesaurus or hyponyms and hypernyms in the ontology. The tag name *year*, however, is not marked (the user is sure of the exact spelling) and will not be expanded. Evaluating this query over the XML data in Fig. 1A, we will get results **4** and **7**. Notice that the term *publication* is automatically expanded to match its hyponyms *book* and *article*.

3.3 Summary

Marking structurally ambiguous elements with the related keyword and ambiguous tag names with the expand function enables a user to query XML documents without perfect knowledge of either the structural relationships among the nodes or the names of these nodes. XQuery equipped with these 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 MQF

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

In the ensuing discussion, for a Schema-Free XQuery with an embedded function mqf($e_1, e_2, \ldots, e_m$), where e_i are the elements involved in the MQF, we use $IList[i] = \{a_{i1}, a_{i2}, \ldots\} \subseteq N$ to represent a list of nodes matching e_i ($1 \leq i \leq m$) in the XML database. The list of MLCASs output by the function is denoted as *OutList*.

4.1 Basic implementation

MQFs can easily be computed using the existing standard query operators. The basic idea is to first find MQFs for each pair of *IList*s. Such MQFs are generated by joining nodes with common ancestors into a set of trees such as the "leaf level" of each tree contains exactly one node from each *IList*; then within each set of the trees generated, we eliminate those whose root node is an ancestor (in the database tree) of the root node of another tree in the set. Next, the MQFs obtained from the previous step are joined with each other based on whether they contain an identical node from $IList[i]$, if they contain leaf nodes from $IList[i]$. The "leaf level" of each tree resulting contains exactly one node from each *IList*. In addition, any pair of leaf nodes in the tree has a non-empty MLCA. These trees are returned as the MQFs.

Theorem 1 *The time complexity of the basic implementation is* $\mathcal{O}(m\mathcal{N}_{\mathcal{T}_m} \lg(\mathcal{N}_{\mathcal{T}_m}) + h\mathcal{M}_{\mathcal{T}_m})$, *where* h *is the height of the XML data tree,* $\mathcal{N}_{\mathcal{T}_m} = \prod_{i=1}^{m} |IList[i]|$ ($m > 1$), *and* $\mathcal{M}_{\mathcal{T}_m} = \sum_{i=2}^{m} \sum_{j=1}^{i-1} |IList[i]| \cdot |IList[j]|$.

Proof The proof is by induction on the value of m. For the base case, let $m = 2$, and $\mathcal{E}_i$ denote the set of nodes in $IList[i]$ ($i = 1, 2$). MQFs of nodes from $IList[1]$ and $IList[2]$ can be computed as $\gamma(o(\upsilon_a(\mathcal{E}_1)) \bowtie_{a-d} \mathcal{E}_2)$ (denoted as $\varpi_{1,2}$ for simplicity), where γ is the duplicate elimination operator, which eliminates a tree based on whether its root is an ancestor (in the database tree) of the root of another tree in the input, o is the sorting operator, υ_a is the ancestor fetching operator, and $\bowtie_{a-d}$ is the ancestor–descendant structural join operator.

The maximum number of ancestors each node in the document tree could have is $h - 1$, for a tree of height h. Therefore, the number of nodes output by $\upsilon_a(\mathcal{E}1)$ is $h|IList[1]|$, given by $|IList[1]|$ plus the total number of ancestors of the nodes in $IList[1]$. Using any typical sorting algorithm such as quick-sort, the time complexity for $o(\upsilon_a(\mathcal{E}_1))$ is $\mathcal{O}(h|IList[1]| \lg(h|IList[1]|))$.

Assuming a stack-based algorithm for the ancestor–descendant structural join, we use nodes from $IList[1]$, and their ancestors, as the ancestor node input list *AList*, while

nodes from *IList*[2] as the descendant node input list *DList*. According to [7], The maximum number of iterations it takes for such a join process of two nodes is $\mathcal{O}(|AList| + |DList| + |TList_{12}|) = \mathcal{O}(h|IList[1]| + |IList[2]| + |TList_{12}|)$, where $TList_{12}$ denotes the output list. In the worst case, nodes from the two *IList*s are all siblings, and thus the number of tree generated is $|TList_{12}| = |AList| \times |DList| = h\prod_{i=1}^{2}(|IList[i]|) = h\mathcal{M}_{\mathcal{T}_2}$. Therefore, the time complexity of the structural join between *IList*[1] and its ancestors with *IList*[2] is $\mathcal{O}(h\mathcal{M}_{\mathcal{T}_2})$.

In stack-based structural join, trees can be output in the order of their respective root nodes without extra cost. During the subsequent duplicate elimination process, each tree will only be visited once and be eliminated at most once. The total time it takes to eliminated unqualified trees from the trees generated by previous step is thus linear to the input size, which is $\mathcal{O}(h\mathcal{M}_{\mathcal{T}_2})$.

Putting all the these together, for $m = 2$, the time complexity is $\mathcal{O}(h\mathcal{M}_{\mathcal{T}_2}) < \mathcal{O}(h\mathcal{M}_{\mathcal{T}_2} + \mathcal{N}_{\mathcal{T}_2}\lg(\mathcal{N}_{\mathcal{T}_2}))$ where $\mathcal{M}_{\mathcal{T}_2} = \mathcal{N}_{\mathcal{T}_2} = |IList[1]| \cdot |IList[2]|$.

Induction Hypothesis: Let the claim hold true for all input with $m < k$.

Induction Step: Let $m =$ k, and $\mathcal{C}_{k-1}$ denote the set of MQFs of input nodes from *IList*[1] to *IList*[$k-1$]. According to the Definition of MQF, to compute the MQF with the kth input *IList*[k], we need to first compute the MQFs of nodes from *IList*[k] with each existing input *IList*[i] using $\varpi_{i,k}$ ($i = 1, \ldots, k-1$). Each list of trees resulting is then joined with $\mathcal{C}_{k-1}$ based on the $startPos$ of their leaf nodes. Such value joins are used to eliminate unqualified trees from the MQFs candidates list: any pair of leaf nodes of an MQF, a_i (from *IList*[i]) and a_j (from *IList*[j]), share a non-empty MLCA, and thus can be found in the result of corresponding $\varpi_{i,j}$. The final MQF candidates are the MQFs needed. Therefore, the MQF with the kth input included is computed as:

$$\mathcal{C}_{k-1} \bowtie_{=1} \varpi_{1,k} \bowtie_{=2,k} \varpi_{2,k} \cdots \bowtie_{=k-1,k} \varpi_{k-1,k}$$

Note that $\mathcal{C}_{k-1}$ contains no node from the kth input, *IList*[k], therefore, the first value join $\bowtie_{=1}$ is only based on the $startPos$ of nodes from *IList*[1]. Each of the MQF candidates resulting contains one nodes from each of the k inputs at the leave level; thus the other value joins $\bowtie_{=i,k}$ ($i = 2, \ldots, k$) are joins on the $startPos$ of nodes from both *IList*[i] and *IList*[k]. The worst-case time for computing $\mathcal{C}_{k-1}$ is $T(k-1) = c_1(k-1)\mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + c_2h\mathcal{M}_{\mathcal{T}_{k-1}}$. The time for first value join $\bowtie_{=1}$ may be up to $|IList[1]| \cdot |IList[k]| + \mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + \mathcal{N}_{\mathcal{T}_k}$, as the output size of $\varpi_{1,k}$ may be up to $|IList[1] \cdot |IList[k]|$, the size of $\mathcal{C}_{k-1}$ may be up to $\mathcal{N}_{\mathcal{T}_{k-1}}$, and the output size of the join may be up to $\mathcal{C}_{k-1} \cdot |List[k]| = \mathcal{N}_{\mathcal{T}_k}$. The time for computing $\varpi_{i,k}$ ($i = 1, \ldots, k-1$) is $h|IList[i]| \cdot |IList[k]|$. The time for each remaining value join $\bowtie_{=i,k}$ ($i = 2, \ldots, k-1$) may be up to $|IList[i]| \cdot |IList[k]| + \mathcal{N}_{\mathcal{T}_k}\lg(\mathcal{N}_{\mathcal{T}_k}) + \mathcal{N}_{\mathcal{T}_k}$, as the size of the MQF candidates after the first value join may be $\mathcal{N}_{\mathcal{T}_k}$.

Therefore, the worst-case time for computing $\mathcal{C}_k$ is

$$\begin{aligned}
T(k) &= c_1(k-1)\mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + c_2h\mathcal{M}_{\mathcal{T}_{k-1}} \\
&\quad + h\textstyle\sum_{i=1}^{k-1}|Ilist[i]| \cdot |IList[k]| \\
&\quad + |IList[1]| \cdot |IList[k]|\mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + \mathcal{N}_{\mathcal{T}_k} \\
&\quad + \textstyle\sum_{i=2}^{k-1}(|IList[i]| \cdot |IList[k]| + \mathcal{N}_{\mathcal{T}_k}\lg(\mathcal{N}_{\mathcal{T}_k}) + \mathcal{N}_{\mathcal{T}_k} \\
&= c_1(k-1)\mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + c_2h\mathcal{M}_{\mathcal{T}_{k-1}} \\
&\quad + (h+1)\textstyle\sum_{i=1}^{k-1}|IList[i]| \cdot |Ilist[k]| \\
&\quad + \mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + (k-2)\mathcal{N}_{\mathcal{T}_k}\lg(\mathcal{N}_{\mathcal{T}_k}) \\
&\quad + (k-1)\mathcal{N}_{\mathcal{T}_k} \\
&\leq c_1k\mathcal{N}_{\mathcal{T}_{k-1}}\lg(\mathcal{N}_{\mathcal{T}_{k-1}}) + c_1k\mathcal{N}_{\mathcal{T}_k}\lg(\mathcal{N}_{\mathcal{T}_k}) \\
&\quad + (k-1)\mathcal{N}_{\mathcal{T}_k} + c_2h\mathcal{M}_{\mathcal{T}_k} \\
&= \mathcal{O}(k\mathcal{N}_{\mathcal{T}_k}\lg(\mathcal{N}_{\mathcal{T}_k}) + h\mathcal{M}_{\mathcal{T}_k})
\end{aligned}$$

Thus the induction step holds. □

4.2 Efficiently computing MQF

Computing MQF using the standard operators, as described above, is simple, but expensive. To efficiently compute MQFs, 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 [7, 8, 11, 15], 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`),[3] 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 MQFs. Within each such tree, the root is an MLCA of the inputs, and the leaf level contains all the nodes from multiple MQFs sharing the same root. Identification numbers are then used to distinguish nodes from different MQFs. 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 Fig. 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. 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. For example, the bottom stack node

[3] `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`).

```
StackNode {
  int maxID;            /*maximum min among all nodes in Elists*/
  NodeType head;        /*a XML tree node*/
  ListNode Elist[m];    /*m: total number of input lists*/
  bitset relBits[l];    /*l: number of groups of Related Elists*/
}
ListNode {
  int min;              /*identification number*/
  NodeType node;        /*a XML tree node*/
}
```

Fig. 6 Data structure of a stack node

in Fig. 8d represents the left branch rooted at node **2** in the document tree A of Fig. 1, with nodes **3**, **5**, and **6** being the leaves.

Some $Elist$s of a stack node may be marked as $Related$ with each other, indicating that the LCA(s) of nodes from these lists are descendants of the head. For example, $Elist[1]$ and $Elist[2]$ of the bottom stack node (**2**) in Fig. 8d are said to be *Related*, because node **4**, the LCA of the elements in the $Elist$s (**5**, in $Elist[1]$, and **6**, in $Elist[2]$), is a descendant of the head node **2**. Meanwhile, $Elist[0]$ is not *Related* to the other two $Elist$s, as the LCA of node **3** (in $Elist[0]$) and any node in the other *ELists* is **2** itself. A bitset array $relBits$, with each element being a vector of bits with a length of m, where each bit corresponds to a $Elist$, is used to keep such information. The size of the array depends on the number of $Related\ Elist$ groups, where $Elist$s within each group are $Related$ to each other, while $Elist$s from different groups are not. Each $Related\ Elist$ group involves at least two $Elist$s, therefore $|relBits| < \lceil \frac{m}{2} \rceil$. In our example, stack node **2** in Fig. 8d has only one $Related\ Elist$ group, $Elist[1]$ (with node **6**) and $Elist[2]$ (with node **5**). Its $relBits$ thus has only one element $relBits[1] = 110$. $Elist[1]$ and $Elist[2]$ are referred to as the $Elist$s corresponding to $relBits[1]$, as the value of the bits representing the two $Elist$s is 1.

4.2.1 Determination of unqualified nodes

We first explain here how to efficiently determine nodes that are unqualified to be part of any MQF. According to Definition 6, if a node $a_i \in A_i$ belongs to an MQF, then the root of the MQF is MLCA($a_1, \ldots, a_i, \ldots, a_m$), where $a_k \in A_k$ ($k = 1, \ldots, m$). Therefore given a node $a_i \in A_i$, if we can quickly determine that we cannot find any $a_k \in A_k$ ($k = 1, \ldots, i-1, i+1, \ldots, m$) such that MLCA($a_1, \ldots, a_i, \ldots, a_m$) exists, we can then conclude that a_i is not qualified to be part of any MQF. Based on Defintion 5, we know that the existence of MLCA($a_1, \ldots, a_i, \ldots, a_m$) depends the existence of MLCA(a_i, a_k) ($k = 1, \ldots, i-1, i+1, \ldots, m$). Therefore, we need only to determine that there exists no MLCA for a_i and any node of another type to conclude that no MLCA($a_1, \ldots, a_i, \ldots, a_m$) exists. We use the following proposition for this purpose.

Proposition 1 (Determine unqualified nodes) *Let the set of nodes in an XML document be N. Given $A_1, \ldots, A_m \subseteq N$, where A_i is comprised of nodes of type $\mathcal{A}_i$ ($i \in [1, \ldots, m]$), $a_j \in A_j$ is said to be unqualified as part of any MQF, if $\exists\ l$, where $l \in [1, \ldots, m]$, $l \neq j$, such that the following conditions hold true:*

- *$\exists\ c \in N$, $c \succeq a_j$, and*
- *$\forall\ a_l \in A_l$, $a_l \not\preceq c$, and*
- *$\exists\ d \in N$, d is a sibling of c in the XML tree, and $d = CA(a'_j, a'_l)(a'_j \in A_j, a'_j \neq a_j, a'_l \in A_l)$.*

In addition, $\forall a_k \in A_k$ ($k \in [1, \ldots, m]$, $k \neq j$), where $MLCA(a_j, a_k) \preceq c$, a_k is also said to be unqualified as part of any MQF, if the above conditions hold true.

Proof We prove the first part of the proposition by contradiction. Suppose that there exists $a_h \in A_h$ ($h = 1, \ldots, j-1, j+1, \ldots, m$) such that MLCA($a_1, \ldots, a_j, \ldots, a_m$) $= e$ ($e \in N$, $e \neq$ ***null***). Then according to Definition 5, there exists $f =$ MLCA(a_j, a_l) ($f \in N$, $f \neq$ ***null***). Condition 2 in the above proposition indicates that c does not have any descendant from A_l, and thus $c \prec f$. Node d is a sibling of c; therefore, $d \prec f$. Since $d =$ CA(a'_j, a'_l), there exists $g \preceq d \preceq f$, and $g =$ LCA(a'_j, a'_l), contradicting $f =$ MLCA(a_j, a_l) (Definition 4). Therefore, a_j is unqualified to be part of any MQF.

The proof of the second part of Proposition 1 is similar. Given an $a_k \in A_k$ ($k \in [1, \ldots, m]$, $k \neq j$), where MLCA(a_j, a_k) $\preceq c$, suppose that there exists $a_h \in A_h$ ($h = 1, \ldots, k-1, k+1, \ldots, m$) such that MLCA($a_1, \ldots, a_k, \ldots, a_m$) $= e$ ($e \in N$, $e \neq$ ***null***). Then according to Definition 5, there exists $f =$ MLCA(a'_j, a_k) ($a'_j \in A_j$, $a'_j \neq a_j$). From Definition 4, we can conclude that $f =$ MLCA(a'_j, a_k) $=$ MLCA(a_j, a_k) $\preceq c$, and thus $a'_j \preceq c$. Then a'_j satisfies all the conditions in Proposition 1, and thus is not qualified to be part of any MQF, contradicting MLCA($a_1, \ldots, a_k, \ldots, a_m$) $= e$ ($e \in N$, $e \neq$ ***null***). Therefore, a_k is not qualified to be part of any MQF. □

For examples of unqualified nodes, consider the document in Fig. 4b. Suppose that we want to find MQFs for nodes of type *title*, *author*, and *review*. We can easily determine that node **2** is not qualified based on Proposition 1: node **1**, the parent of node **2**, does not have descendants of type *author*, but its sibling **5** is MLCA(**6**,**7**), where node **6** is of type *title* (same as **2**), and node **7** is of type *author*. Therefore, no MLCA of node **2** and an *author* node exists; consequently, node **2** can not be part of any MQF involving both *title* and *author*. Similarly, we can determine that node **4**, a sibling of node **2**, is also unqualified.

Now, we show how the concept of $Related$ can be used to take advantage of Proposition 1 and eliminate unqualified nodes in the $Elist$s. Given two stack nodes s_1, s_2, where $s_1.head$ is the parent of $s_2.head$, and the $Elist$s of s_1 contains all the descendants of $s_1.head$ before $s_2.head$ in the XML tree, while $Elist$s of s_2 contains all the descendants of $s_2.head$ in the XML tree, we can easily determine which $Elist$s of the two nodes contains unqualified nodes. For i, j ($i \neq j$), if $s_1.Elist[i]$ and $s_1.Elist[j]$ are not $Related$, then for any a_{i1} from $s_1.Elist[i]$, we can find an ancestor node c_1 for a_{i1}, where c_1 is a child of $s_1.head$, and c_1 has no descendant of type $\mathcal{A}_j$. If $s_2.Elist[i]$ and $s_2.Elist[j]$ are $Related$,

then for any a_{i2} from $s_2.Elist[i]$, and a_{j2} from $s_2.Elist[j]$, $s_2.head = CA(a_{i2}, a_{j2})$. Since $s_2.head$ is a sibling of c_1, we can now apply Proposition 1 and determine that nodes from $s_1.Elist[i]$ and other *Related Elists* are unqualified as part of an MQF and thus can then be deleted. The bitset array $relBits$ can be used to efficiently determine whether Proposition 1 can be applied by simple logical operations. For example, given $s_1.relBits[1] = 0111$, $s_2.relbits[1] = 0110$, since

$$s_1.relBits[1] \& s_2.relbits[1] = 0110 = s_2.relbits[1],$$

and

$$s_1.relBits[1] | s_2.relbits[1] = 0111 = s_1.relBits[1],$$

we can quickly determine that $Elist[2]$ are *Related* to $Elist[4]$ in s_1, but not in s_2; therefore nodes in the $Elists$ ($Elist[2]$ and $Elist[3]$) corresponding to $s_2.relBits[1]$ are not qualified as part of an MQF.

4.2.2 Full algorithm

The full algorithm is shown in Fig. 7. The algorithm proceeds as follows: First, if the stack is empty, or the head of the current stack top is an ancestor of the current input node t, t is directly pushed onto the stack (`MQF` line 7, `PopStack` line 12). Otherwise, we first check if the current stack top already contains MQFs: if it does, we output current stack top, empty the stack (`PopStack` lines 3–4), and push t onto the stack (`MQF` line 7); else, we repeatedly either replace the head of the current stack top with its parent (`PopStack` lines 7–8), or insert the $Elists$ of stack top into the node under it (`PopStack` lines 9–11), until we find t is a descendant of the new stack top. In addition, before we insert $Elists$ of one stack node into another, unqualified nodes in both stack nodes involved can be found and discarded based on Proposition 1 (`DeleteUnqualifiedNodes` lines 1–9). Then, we push t onto the stack. The above process reiterates until we have already processed all the input nodes of at least one *IList*, and the stack is empty.

We now walk through the algorithm using an example. Consider the XML document in Fig. 1A and Query 1 in Fig. 5. For the function mqf($\$a$, $\$t$, $\$y$), the input lists are $IList[1] = \{\mathbf{6}, \mathbf{9}, \mathbf{10}, \mathbf{15}, \mathbf{18}\}$, $IList[2] = \{\mathbf{5}, \mathbf{8}, \mathbf{14}, \mathbf{17}\}$, $IList[3] = \{\mathbf{3}, \mathbf{12}\}$, matching elements *author*, *title*, and *year*, 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 (`MQF` line 7) (see Fig. 8a for illustration). In the following discussion, for the seek of simplicity, we use the id of the head of a stack node to distinguish different stack node (e.g., stack node **3** refers to the stack node whose head is **3**), as no two nodes on the stack will have the same head node at any time.

The algorithm then reads in the next element with the smallest `StartPos`, **5** (a *title*), which is not a descendant of the head of the stack top. Since the $Elists$ of stack node **3** are all empty, and no unprocessed descendant node of **3** remains, it is guaranteed that no MQF rooted at **3** can be

```
MQF (I_1, I_2, ..., I_m):
1 let the set of input nodes from I_1, I_2, ..., I_m be I
2 while ∀i, I_i is not empty; or stack is not empty
3     do let t_min(from I_k) be the node with smallest StartPos
4           among unprocessed nodes in I
5        result ← Popstack(t_min)
6        if result ≠ null, then output result
7        PushNewStackNode(t_min, k)

Popstack(t):
1 while stack is not empty and t is not a descendant of stack top
2  do popped ← stack.Pop(), top ← stack.Top()
3     if popped and its Elists contain MQFs
4        then empty stack, return popped
5     else mark all the non-empty Elists of popped as Related
6           in popped.relBits[1]
7        if popped.head is a child of top.head
8              MergeElists(popped, top)
9        else pt ← popped.head.GetParent()
10             popped.head ← pt
11             stack.Push(popped)
12 return null

MergeElists(s, t):
1 if DelUnqualifiedNodes(s, t)
   /* append s.EList and update t.relBits accordingly */
2     then t.AppendLists(s.GetLists())
3          t.maxID ← s.minID

DeleteUnqualifiedNodes(s, t):
1 for i ← 1 to ⌈m/2⌉
2  do if t.relBit[i].count() > 0
3     then if t.relBits[i] = s.relBits[1] then return true
4        else if t.relBits[i]&s.relBits[1].count() > 0
5           then relBitsOR ← (t.relBits[i])|(s.relBits[1])
6              if t.relBits[i] = relBitsOR then return false
7              else delete t.Elists corresponding to relBitsOR
8                 if s.relBits[i] = relBitsOR then return true
9                 else return false
10    else return true

PushNewStackNode(t, k):
1 if stack is empty
2     then stack.Push(t)
3          top ← stack.Top()
4          top.minID ← 0, top.maxID ← 0
5 else oldtop ← stack.Top()
6        stack.Push(t)
7        top ← stack.Top()
         /* assign min to differentiate nodes of MQFs */
8        if oldtop.Elist[k] = null
9           then top.minID ← oldtop.minID
10       else if oldtop.Elist[k] is not Related to other oldtop.Elists
11             then top.minID ← oldtop.minID
12          else top.minID ← oldtop.maxID + 1
   /* create a new ListNode with min of top.min */
13 node ← NewListNode(t, top.min)
14 top.Elist[k].AppendNode(node)
```

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

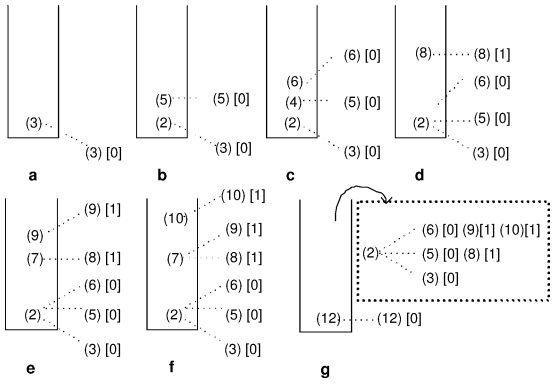


Fig. 8 States of stack during the evaluation of mqf($\$a, \$t, \$y$). Each square bracket contains the *min* value used to distinguish nodes from different MQFs

found. The head of the current stack top **3** is therefore replaced with its parent **2** (`PopStack` lines 9–11). Element **5** is a descendant of node **2**, and is thus pushed onto the stack (Fig. 8b). Similarly, when **6** is read in, we replace the head of stack node **5** with its parent **4**, and then push **6** onto the stack (Fig. 8c). Note that this is a subtle, yet important, optimization to the algorithm: we access an ancestor node only when it is needed to compute MQFs.

Element **8** is read in next and it is again not a descendant of the stack top **6**. However, node **6** is found to be a child (not just descendant) of node **4**, the head of the stack node below it (`PopStack` line 7). Therefore the stack top and its *Elist*s are recursively appended to the stack node below it (`MergeElists` lines 1–2).[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 (Fig. 8d). Note that the *min* value assigned to **8** is different from that of **5** (`PushNewStackNode` lines 10–12). The meaning and usage of *min* will be discussed soon.

The process of adding **9** and **10** is similar to that of adding **5** and **6** (Fig. 8e and f). When **12** is read in, as what happens when element **8** is read in (Fig. 8d), 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 MQFs), and popped as output. It is guaranteed that all the MQFs sharing **2** as the root have been found (in the *Elist*s), since all the descendants of **2** in the input have been processed. We then push **12** onto the empty stack (Fig. 8g). The algorithm continues until one of the *IList*s is empty and the stack is empty.

Identification numbers $[min, max]$ are used to distinguish different MQFs. *min* is assigned for each input element when it is added to the stack (`PushNewStackNode`), 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 MQF(s), while element from not *Related Elist*s may belong to the same MQF(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 MQF and construct MQFs.

[4] First insert the *Elist*s of **6** into **4**; then insert the *Elist*s of **4** into **2**.

Theorem 2 *The time complexity of the stack-based MQF algorithm is* $\mathcal{O}(h \sum_{i=1}^{m} |IList[i]| + |OutputList|)$*, where OutputList refers to the MQFs generated.*

Proof We start by showing that it takes $\mathcal{O}(h\sum_{i=1}^{m} |IList[i]|)$ time to compute structures containing MQFs with the same root. During the computation, each input node will be pushed onto the stack once. Ancestor nodes of each input node may also be pushed onto the stack, until the node is determined to be unqualified to be a part of any MQF, or MQFs containing the node have already been found. In the worse case, the only common ancestor node of all the input nodes is the root of the document. In such a case, all the ancestor nodes of each input node will be pushed onto the stack, even though the same ancestor node of multiple input nodes will be pushed onto the stack only once. Each stack node may have up to $h - 1$ ancestor nodes, but the root node will be pushed onto the stack only once; therefore, the total number of nodes pushed onto the stack is $(h - 2) \sum_{i=1}^{m} (|IList[i]|) + 1 = \mathcal{O}(h \sum_{i=1}^{m} |IList[i]|)$. Each node pushed onto the stack may be popped from the 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; thus appending and deletion of a single node in the *Elist*s can be performed in unit time). Since each of the stack operations requires constant time, the time complexity of computing the structures containing MQFs with the same root is $\mathcal{O}(h \sum_{i=1}^{m} |IList[i]|)$, which proves the claim.

To complete the proof of the theorem, let l be the number of structures generated from previous step, and $|OutputList|$ be the number of MQFs generated. The process to generate MQFs from the structures generated from previous step is essentially a sort merge join, where MQFs are generated from the join of compatible nodes from different *EList*s within each structure based on their (*minID*, *maxID*). Since nodes in each *EList* are already sorted by (*minID*, *maxID*), the time required for this merge process is linear to the sum of input and output size, which is $\sum_{i=1}^{m} \sum_{j=1}^{l} |EList_j[i]| + |OutputList|$. Since each input node may only appear in at most one *EList*, we can obtain $\sum_{j=1}^{l} |EList_j[i]| \leq |IList[i]|$; thus, $\sum_{i=1}^{m} \sum_{j=1}^{l} |EList_j[i]| \leq \sum_{i=1}^{m} |IList[i]|$. Putting this observation together with the observations above, the time complexity of the stack-based MQF algorithm is $\mathcal{O}(h \sum_{i=1}^{m} |IList[i]|) + \mathcal{O}(\sum_{i=1}^{m} \sum_{j=1}^{l} |EList_j[i]| + |OutputList|) = \mathcal{O}(h \sum_{i=1}^{m} |IList[i]| + |OutputList|)$. □

5 Experimental evaluation

We implemented Schema-Free XQuery using Timber [2, 27], a native XML database, and evaluated the system on two aspects: (1) search quality, and (2) search performance. Search quality is evaluated using both a standard XML benchmark (Sect. 5.1) and a heterogeneous data collection (Sect. 5.2). For search performance, we measure the overhead caused by evaluating schema-free query versus the schema-aware query (Sect. 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.[5] 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, it is not reasonable to establish a precision–recall curve for our search quality evaluation.

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 [4]. 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 106 MB 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 [34] and XSEarch [17]. Meet proposes to find the LCA for the set of keywords given in the query and returns 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).

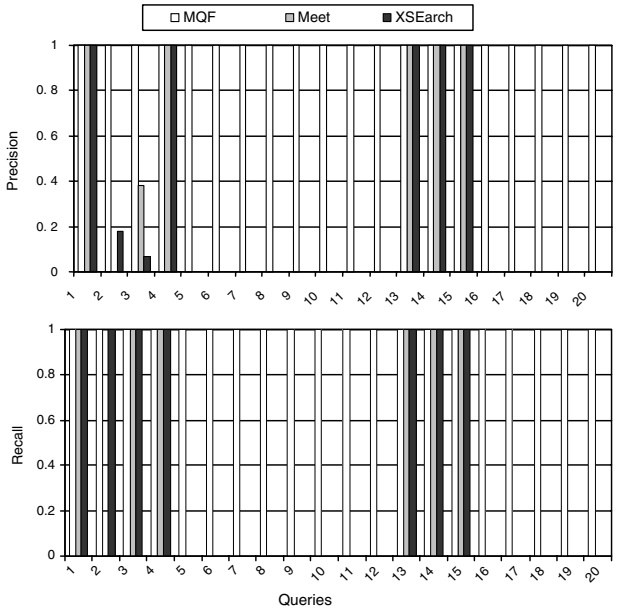


Fig. 9 Precision and recall of different search strategies on XMark. Missing bars indicate a value of zero

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. The query numbers shown along the X-axis are the numbers used to identify queries in XMark. Schema-Free XQuery (MQF) achieved perfect precision and recall for all the queries (i.e., all the results returned by mqf-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 Fig. 9): unrelated elements are returned along with the meaningful ones.

[5] Query answers are not discrete documents, as in standard IR, but rather fragments of XML. So answers that return the required information, but an unnecessarily large fragment, are still correct, though they may not be specific. See extensive discussions of this issue in INEX[1]. For our experiments, we only considered correctness and not specificity – see the last paragraph of Sect. 5.1 for a discussion of the consequences of our choice.

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 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 [36], were largely based on bibliography documents, which were relatively easy to collect from the web. We therefore decided to use the 11 queries[6] from "XMP," plus an example query (also based on bibliography data) from XSEarch [17] for this part of evaluation. (This 12th query is one chosen by the XSearch authors to highlight their system, and hence is likely to be one that most strongly tests any system being compared against theirs).

Publication collection We manually collected personal publication lists from 300 faculty personal home pages in a large research university[7] to serve as the data set for the "XMP" queries. The publication lists, while obtained from the real world, are semantically close enough to the bibliography data 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[8] 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 natural language query, we ran exactly the same Schema-Free XQuery on all the publication lists.

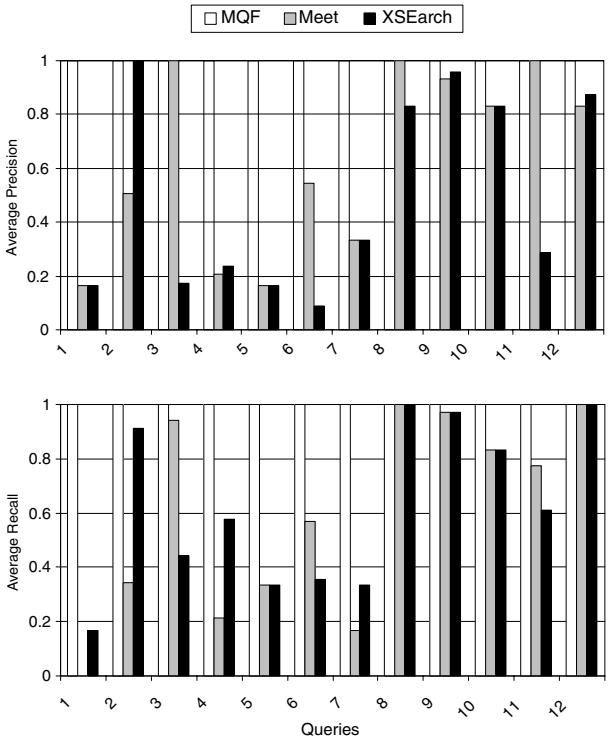


Fig. 10 Average precision/recall of different search strategies for publication collection with term expansion. Missing bars indicate a value of zero

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 data set and/or any query. For instance, if we change the XML document shown in Fig. 1A such that *author* node **6** and *title* node **8** are removed, for Query 1 in Fig. 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 (Fig. 11). It is not a surprise to see that a mismatch on even one single tag name could reduce the search quality significantly.

5.3 Search performance: mqf-embedded XQuery

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 mqf-embedded XQuery and the corresponding XQuery, with mqf computation being considered a single

[6] Q12 of XMP is not included since set comparison is not yet supported in Timber. Instead, we used the example proposed in XSearch.

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

[8] Over the six representative documents.

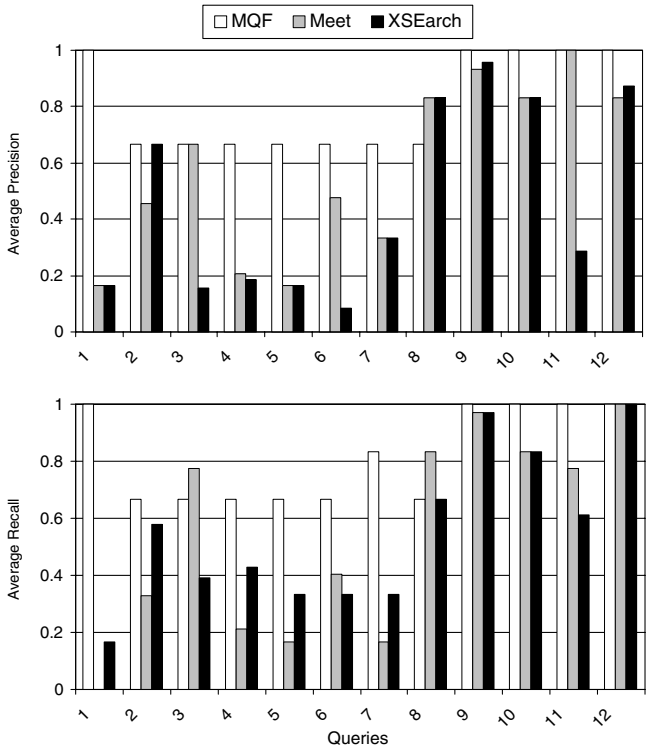


Fig. 11 Average precision/recall of different search techniques for publication collection without term expansion

operator. To evaluate efficiency, we compare the time cost of evaluating an mqf-embedded XQuery, with both the basic and stack-based implementations, 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 [28], which was of sufficient size to show non-trivial running 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, 512 MB RAM, 120 GB hard disk) running Windows 2000 Professional. The Timber buffer size was set to 64 KB. 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 MQF previously discussed in Sect. 4.1), the execution time for some queries is marked as DNF (did not finish), which means that the execution was killed when it did not finish within 7 h.

Results Figure 12 shows that evaluation plans generated by mqf-embedded XQuery for the "XMP" queries on DBLP data are usually simpler than those of XQuery, with two fewer operators on average, an approximately 25% savings in plan generation. Furthermore, unlike the schema-aware

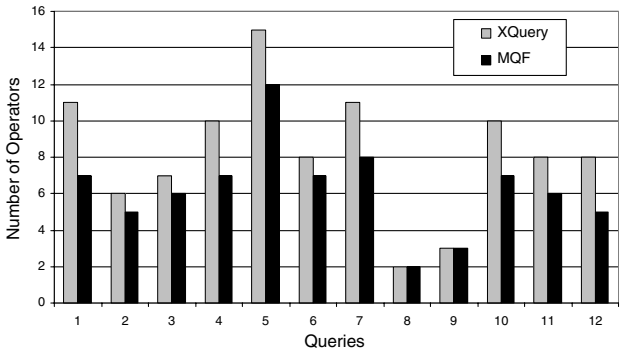


Fig. 12 Average number of operators in evaluation plans generated by schema-aware XQuery (XQuery) and mqf-embedded XQuery (MQF) for "XMP" queries on DBLP

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 mqf-embedded XQuery, both the stack-based algorithm (MQF) and the basic algorithm (COMPOSE), and schema-aware XQuery (XQuery) for the "XMP" queries on DBLP data. Our stack-based MQF algorithm expedites the processing of mqf-embedded XQuery by approximately 16 times, often reducing the execution time from more than 7 h to less than 30 min. The capability of Schema-Free XQuery does not incur expensive performance cost. The overhead of mqf-embedded XQuery using MQF 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 MQFs is needed. The existence of such overhead is expected. mqf-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 MQFs, while in schema-aware XQuery, most such filtering is done at data fetching time. In Sects. 6 and 7, 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 mqf-embedded XQuery is 26.2 s, while that of schema-aware XQuery is 12.3 s. We cannot compute the geometric mean of the running time for COMPOSE, as 11 out of 20 queries failed to finish within 7 h. The overhead for mqf-embedded XQuery, compared to schema-aware XQuery varied from 0 to 250%.

6 Integrating MQF computation into a query evaluation pipeline

In this section, we first introduce a motivating example on the integration of MQF computation into a query evaluation pipeline. The concept of MLCA is then slightly modified to fit into this context (Sect. 6.1). In Sect. 6.2, an *Ancestor–Descendant Summarization* (A–D) Index is proposed to facilitate the MQF computation within a query evaluation pipeline.

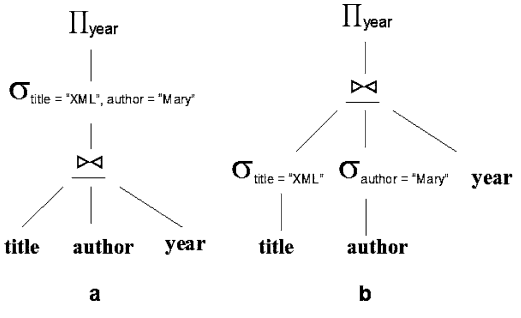
Table 1 Performance (seconds) of XQuery, MQF, and COMPOSE for the "XMP" queries on DBLP

Query	XQuery	MQF	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

6.1 Motivation

As we have previously discussed, the evaluation overhead of a schema-free query is mainly because MQF operators are at the leaf of query evaluation plans, and thus results filtering according to search conditions is always done after the computation of MQFs. An intuitive heuristic to improve the query evaluation plan of mqf-embedded XQuery is to reduce the size of the inputs of MQF operators by applying selection early. However, simply pushing selection operators ahead of MQF operators may result in semantically incorrect MQFs.

Consider Query 4 in Fig. 13, posed against data in Fig. 1A. If we apply the straightforward evaluation plan in Fig. 13a, there will be no MQF satisfying the selection conditions. If we apply selection before the MQF operator as shown in the alternative evaluation plan in Fig. 13b, the input size of the MQF operator is reduced from 11 nodes to 4 nodes, with only *title*{**5**}, *author*{**10**}, and *year*{**3**,**12**}, rather than all the *title*, *author*, and *year* nodes. However, the MQF generated by algorithm MQF, {**2**,**5**,**10**,**3**}, and the result returned, *year* = "1999", are incorrect: *title* {**5**} and *author*{**10**} belong to different publications and are not meaningfully related to each other.



Query 4: Find year of the publications with title XML, of which Mary is an author.

Fig. 13 Example Evaluation Plans. ⋈ denotes MQF operation

As we can see from the above example, to provide the strict semantic guarantee, we cannot simply push selections before an MQF operator and completely ignore those input nodes eliminated by early selections: such nodes may help to determine the correct MQFs. When we integrate MQF into a query pipeline, and allow operations such as selections be done before MQF operation, we need to depend not only on the current input nodes of the MQF operator, but also on the nodes in the original database that are not part of the inputs but are of the same types as the input nodes. For example, during the MQF computation using evaluation plan **b** in Fig. 13, if we check the base data in Fig. 1, we can find that node **4**, the LCA of node **5** and another *author* node **6**, is lower than node **2**. According to Definition 6, we can immediately determine that {**2**,**5**,**10**,**3**} is not an MQF.

We capture the above intuition by extend the MLCA definition of two nodes as following. The definitions for MLCA of multiple nodes and MQF remain unchanged.

Definition 8 (MLCA of two nodes) Let the set of nodes in an XML document be N. Given A', $B' \subseteq N$, where A' comprises all the nodes of type $\mathcal{A}$ in N, and B' comprise all the nodes of type $\mathcal{B}$ in N, the Meaningful Lowest Common Ancestors Set $C \subseteq N$ of A and B ($A \subseteq A'$, $B \subseteq B'$) satisfies the following conditions:

- $\forall\, c \in C, \exists\, a \in A, b \in B$, such that $c_k = \text{LCA}(a,b)$. c_k is denoted as $\text{MLCA}(a, b)$.
- $\forall\, a \in A, b \in B$, if $d = \text{LCA}(a, b)$ and $d \notin C$, then $\exists\, a' \in A', b' \in B'$, such that $d' = \text{LCA}(a', b')$ and $d' \prec d$.

The set C is denoted as $\text{MLCASET}(A, B)$.

Note that in the second condition of the above definition, if $d = \text{LCA}(a, b)$ is an ancestor of $d' = \text{LCA}(a', b')$, then both $a' \prec d$ and $b' \prec d$ hold true. Since any $\text{CA}(a', b') \succeq \text{LCA}(a', b')$, we can obtain the following proposition, with A, B, N remaining the same as previously defined:

Proposition 2 (Determination of MLCA of two nodes) *Given any $d = LCA(a, b)$, where $a \in A$, $b \in B$, d is $MLCA(a, b)$ unless $\exists\, d'' \in N$, $d'' \prec d$, and $d'' = CA(a'', b'')$, where $a'' \in A''$, A'' is comprised of all the type $\mathcal{A}$ descendant nodes of d, and $b'' \in B''$, B'' is comprised of all the type $\mathcal{B}$ descendant nodes of d.*

6.2 Ancestor–Descendant summarization index

According to Proposition 2, to determine whether a node $d = \text{LCA}(a, b)$ is a MLCA, we only need to know the existence of $d'' = \text{CA}(a'', b'')$; we do not need to know exactly which node a'' or b'' is. Therefore, given a node $d = \text{LCA}(a, b)$, if we can quickly determine the existence of $d'' = \text{CA}(a'', b'')$, where $d'' \prec d$, we can immediately conclude that d is not $\text{MLCA}(a, b)$. To take advantage of this property to facilitate MQF computation, we propose the following *Ancestor–Descendant Summarization* (A–D) index structure to enable fast detection of the existence of d''s.

A data entry of an A–D index consists of a pair ⟨`StartPos`, `descendant map`⟩, where `StartPos` identifies a node in the XML document, and the `descendant map` is a fixed number of bits representing the types of its descendant nodes. Each bit of the map represents a particular type: the value of a bit being "1" indicates that the node has a descendant node of that type. Information on node types and the corresponding positions for each node type in a descendant map is stored as a separate index called *Type–Position* index, with each data entry consisting of a pair ⟨`type`, `map position`⟩.

For example, in the A–D index of document Fig. 1A, the entry for node **1** is ⟨0, 0111111⟩, indicating that the `StartPos` of node **1** is 0, and its descendants contain all types of nodes, except the *bibliography* type (position 0). The entry in Type–Position index for the type of *bibliography* is ⟨*bibliography*, 0⟩, i.e., the bit at position 0 in the descendant map of a node indicates whether that node has descendants of type *bibliography*.

By employing A–D index, given any document node, we can quickly determine whether this node is a common ancestor of nodes of two different types via a single index access.[9] A node $e \in N$ is denoted as CA($\mathcal{A}$,$\mathcal{B}$), if it is a common ancestor of nodes of type $\mathcal{A}$ and $\mathcal{B}$. Clearly, no ancestor node of e could be an MLCA of nodes of type $\mathcal{A}$ and $\mathcal{B}$. Based on this notion, we propose the following proposition to efficiently determine the MLCA of two nodes by applying the A–D index.

Proposition 3 (Index-based Determination of MLCA of Two Nodes) *$\forall\, d \in N$, $d = LCA(a, b)$, where a is of type $\mathcal{A}$, and b is of type $\mathcal{B}$, d is MLCA(a, b), unless $\exists\, e \in N, e \prec d$, and $e = CA(\mathcal{A},\mathcal{B})$.*

The size complexity of A–D index is $\mathcal{O}(|N|{\cdot}T)$, where $|N|$ is the size of the XML document tree and T is the total number of node types in the document. We expect that typically T is a small number compared to $|N|$, and thus the size of such A–D index is $\mathcal{O}(|N|)$, approximately proportional to the document size.

The time complexity of an A–D index construction is $\mathcal{O}(|N|)$ since we only need to traverse the entire XML documents once (in depth-first order) to build the index. The time complexity of the construction of the Type–Position index is $\mathcal{O}(T)$, which may be regarded as a constant with respect to the size of the document.

Note that by using the above A–D index to summarize the types of descendants of a node, we assume that nodes of the same tag name correspond to the same type of real-world entities. Sometimes nodes of the same tag names may represent different types of real-world entities within the same schema depending on context. In such cases, A–D index will not be able to distinguish them, unless we know exactly which node corresponds to which type of real-world entity and build the A–D index accordingly. For example, in Fig. 14, node **2** and **4** represent different types of real-world entities, even though they are of the same tag name "id". However, they are indistinguishable in the A–D index entry for node **1** ⟨0, 0111⟩, as they both correspond to the same bit in the descendant map 0111. In general, such cases are very difficult, if not impossible, to be optimized by applying selection early, since we need to depend on the base data to distinguish the nodes of the same tag name but different semantic types.

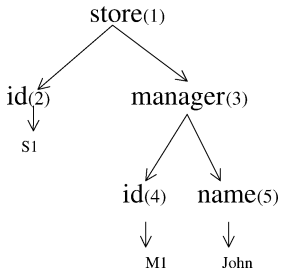


Fig. 14 Same tag name represents different types

[9] Type–Position index only needs to be accessed once initially.

7 Efficiently computing MQF within a query pipeline

We present two different algorithms to utilize A–D index for efficient MQF computation within a query pipeline: MQF Plus A–D Index (MQF+), and Selectivity-based MQF (SQF). Both MQF+ and SQF rely on A–D index to check base data during computation. In addition, SQF uses a selectivity estimator to decide the processing order of inputs. We assume that the map position of each type is known by accessing the Type–Position index initially.

7.1 MQF+

The primary difference between the algorithm MQF+ and MQF is that to determine whether a node may be qualified to be a part of an MQF, the algorithm MQF+ also needs to consider base data not included in the inputs by accessing A–D index. Specifically, given a stack node as described in Sect. 4, if the current input is not a descendant of current stack top, we need to check the base data to determine whether the nodes in the $Elist$s of the current stack top does not qualify to be a part of MQF based on Proposition 3, before we proceed to update the stack. If the A–D index indicates that the head of the stack node c has descendants of type $\mathcal{A}_i$ in the base data, but the $Elist$s of the stack node contains no node of type $\mathcal{A}_i$, it is guaranteed that no nodes in the current stack top is part of an MQF, and the stack node thus can be discarded. The only function being affected in MQF is `Popstack()`, whose updated version is shown in Fig. 15, with new lines 7–10; all the other functions remain the same.

We now walk through the MQF+ algorithm using an example. Consider the XML document Fig. 1A and Query 1 in Fig. 5. For the function mqf($\$a$, $\$t$, $\$y$), if we push the selection $\$a$ = *"Mary"* before the MQF operator, the input lists

```
Popstack(t):
1 while stack is not empty and t is not a descendant of stack top
2  do popped ← stack.Pop(), top ← stack.Top()
3     if popped and its Elists contain MQFs
4        then empty stack return popped
5     else
6        for i ← 1 to m
7           do if Check[i] = true
8              if popped.Elist[i] = null and popped.desc[i] = true
9                 then popped ← null
10                     break
11       if popped ≠ null
12       then mark all the non-empty Elists of popped as Related
13            in popped.relBits[1]
14          if popped.head is a child of top.head
15             then MergeElists(popped, top)
16          else pt ← popped.head.GetParent()
17               popped.head ← pt
18               stack.Push(popped)
19 return null
```

Fig. 15 New `Popstack` function for algorithm MQF+. *Check* is an array storing flag indicating whether early selections have been applied to reduce the input size of *IList*[*i*]; *popped.desc* (line 8) is the descendant map of *popped.head*, obtained by checking A–D index

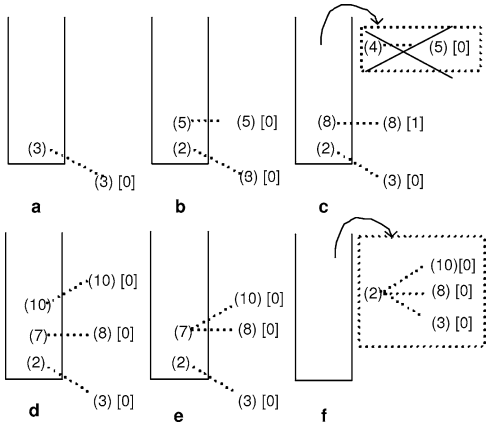


Fig. 16 States of stack during the evaluation of mqf($a, $t, $y) using algorithm MQF+. Each square bracket contains the *min* value used to distinguish nodes from different MQFs

are *IList*[1] = {**10**}, *IList*[2] = {**5**, **8**, **14**, **17**}, and *IList*[3] = {**3**, **12**}, matching elements *author*, *title*, and *year*, respectively. 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 (MQF lines 3–7) (see Fig. 16a for illustration).

The algorithm then reads in the next element with the smallest `StartPos`, **5** (a *title*), which is not a descendant of stack top **3**. The head of the current stack top, element **3**, is then replaced with its parent **2** (`PopStack` lines 16–18). Now **5** is a descendant of new stack top **2**, and is thus pushed onto the stack (Fig. 16b).

Similarly, when **8** is read in, we replace **5** with its parent **4**. Node **8** is not a descendant of **4** either, but before any further manipulation of stack node **4**, we find that node **4** has descendant(s) of type *author* by checking the A–D index, also we find no nodes of type *author* in its *Elists*(`PopStack` lines 6–10). As we have discussed earlier, no nodes in such stack node is qualified for MQF, and thus node **4** is discarded. Element **8** is now a descendant of new stack top **2**, and is pushed onto the stack (Fig. 16c).

Element **10** is read in next, and it is again not a descendant of the stack top **8**. Similarly, we replace **8** with its parent **7**. Element **10** is now a descendant of new stack top **7**, and is pushed onto the stack (Fig. 16d).

When **12** is read in, the current stack top **10** is not an ancestor of **12**, but it is a child of **7**, the stack node right below it. Therefore the *Elists* of **10** are merged into **7** (`PopStack` lines 14–15) (Fig. 16e). Similarly, the *Elists* of **7** are merged into **2**, the stack node below it. The new stack top, **2**, is not an ancestor of **12** either, but it is found to contain no empty *Elists* (indicating that it contains MQFs), and thus popped as output. It is guaranteed that all the MQFs sharing **2** as the root have been found (in the *Elists*). The algorithm stops here, as both the input list of *author* and the stack are empty.

Theorem 3 *The time complexity of the stack-based MQF+ algorithm is* $\mathcal{O}(mh\sum_{i=1}^{m}|IList[i]| + |OutputList|)$*, where m denotes the number of input lists, and h denotes the height of the XML data tree.*

In MQF, the time complexity of stack operation is $\mathcal{O}(h\sum_{i=1}^{m}|IList[i]|)$. The new `Popstack` function in MQF+ adds up to a factor of m extra cost for each stack operation. Therefore, the time complexity of stack operation in MQF+ is $\mathcal{O}(mh\sum_{i=1}^{m}|IList[i]|)$. The time required for merging MQFs from the output trees is still linear in the sum of the input and output size. Thus, we obtain a time complexity of $\mathcal{O}(mh\sum_{i=1}^{m}|IList[i]| + |OutputList|)$ for MQF+. Although the time complexity of MQF+ is worse than that of MQF, because the number of inputs m is usually small, we expect that the actual performance of algorithm MQF+ will be better than algorithm MQF, when the input size of MQF operator is greatly reduced via its integration into the query evaluation pipeline.

7.2 SQF

Both algorithms MQF and MQF+ scan all the input nodes in the order of their `StartPos`. When there exist significant differences among the selectivity of the input lists, nodes from inputs with low selectivity are more likely to determine the MQFs in the result. For example, for query "Find years for all the articles published by author Mary", the number of MQFs generated is bound by the number of *author* nodes with the value of "Mary." Intuitively by starting the computation of MQFs from input nodes with the lowest selectivity, we reduce the computational cost resulted from input nodes that do not belong to any MQF. Based on this intuition, we propose the algorithm SQF.

The pseudo code for SQF is given in Fig. 17. The algorithm SQF is essentially the same as algorithm MQF+,

```
SQF (I_1, I_2, ..., I_m):
1 let the set of input nodes from I_1, I_2, ..., I_m be I
2 while ∀i, I_i is not empty; or stack is not empty
3     do k ← FindInput()
4        if k ≠ -1
5           then t ← I_k.next()
6                result ← Popstack(t)
7                if result ≠ null, then output result
8           PushNewStackNode(t, k)
9        else t ← doc.root
10               result ← Popstack(t)
11               if result ≠ null, then output result

FindInput():
1 if stack is empty
2     if I_S[1] = null then return -1
3     else return S[1]
4 top ← stack.Top()
5 for i ← 1 to m
6  do if top.desc[i] = true
7     then t ← I_S[i].next()
8        while t.StartPos < top.StartPos
9           do t ← I_S[i].next()
10       if t.StartPos > top.EndPos or t = null
11          then if top.Elist[i] = false
12                  then stack.Pop()
13                       return FindInput()
14       else return S[i]
15 if top and its Elists contain MQFs
16    then return -1
17 popped ← stack.Pop(), top ← stack.Top()
18 mark all the non-empty Elists of popped as Related
19    in popped.relBits[1]
20 if popped.head is a child of top.head
21    then MergeElists(popped, top)
22 else pt ← popped.head.GetParent()
23       popped.head ← pt
24       stack.Push(popped)
25 return FindInput()
```

Fig. 17 Algorithm SQF. Function `PopStack` and `PushNewStackNode` are the same as the ones used in algorithm MQF. Array S stores the index of the input lists in the ascending order of their selectivity; $top.desc$ (`FindInput` line 6) is the descendant map of $top.head$, obtained from A–D index.

except it uses the `FindInput` function to decide which input to read in. The read-in order of an input not only depends on its `StartPos`, but also depends on its selectivity. The information on the selectivity of each input is provided by a selectivity estimator prior to query evaluation and stored in an array S. In addition, base data are also checked to discard unqualified stack nodes within the `FindInput` function (we thus use `PopStack` function of MQF, rather than that of MQF+).

The `FindInput` function proceeds as follows. First, if the stack is empty, the input with the lowest selectivity is chosen (line 1–3). Otherwise, the inputs that are the descendant of the head of the current stack top are read in the order of their selectivity (lines 5–14). If such input can be found, and the $Elist$s of the stack top does not contain any nodes from the input lists, it is guaranteed that the stack top does not qualify to be a part of MQF and is thus discarded (lines 10–13) before we continue searching for the next input. If

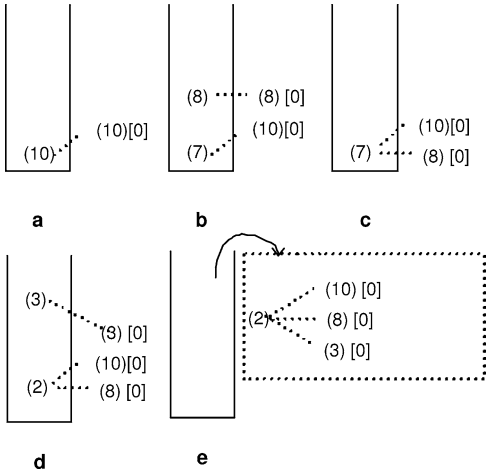


Fig. 18 States of stack during the evaluation of mqf($a, $t, $y) using algorithm SQF. Each square bracket contains the min value used to distinguish nodes from different MQFs

no input can be found for current stack top, we try to output all the MQF on the stack (lines 15–16). Otherwise, we try to either replace the head of the current stack top with its parent, or insert the $Elist$s of stack top into the node under it, similar to what we do in `PopStack` (lines 20–24).

The states of the stack are illustrated in Fig. 18 for the same example as the one used for MQF+ in Fig. 16. There are two major differences between SQF and MQF+ affect the states of stack. First, in SQF, the input read in order is **10**, **5**, **8**, **3**, depending on both selectivity and `StartPos` (Fig. 18), while for MQF+ in Fig. 16, the read in order is **3**, **5**, **8**, **10**, solely determined by `StartPos`. Second, in SQF, unqualified inputs may be ignored without stack operations. For example, in Fig. 18b, element **8** is directly pushed onto the stack, while element **5** is skipped without stack operation (`FindInput` lines 8–9).

Theorem 4 *The time complexity of the SQF algorithm is* $\mathcal{O}(mh\sum_{i=1}^{m}|IList[i]| + |OutputList|)$*, where m denotes the number of input lists, and h denotes the height of the XML data tree.*

The time complexity of SQF is the same as MQF+, as it also adds up to extra m unit operation cost for each node pushed onto the stack. Again, the time complexity of SQF is worse than that of MQF. Because the number of inputs m is usually small, and unqualified nodes can be discarded without stack operations, we still expect that when the input size of MQF operation is reduced via integration into the query evaluation pipeline, SQF will generally perform better than MQF; we also expect that when there are significant differences among the input sizes, SQF will perform better than MQF+, since fewer stack operations may be needed.

8 Experimental evaluation of MQF+ and SQF

To evaluate the benefit of algorithm MQF+ and SQF over the original stack-based MQF algorithm, we compared the

time cost of evaluating mqf-embedded XQuery using the different implementations of MQF operator. The results are reported in Sect. 8.1. We then further examined the impact of selection conditions and input size on the performance of the different implementations of MQF operator in Sect. 8.2.

All the experiments were carried out on the same machine used for previous experiments described in Sect. 5.3, with the same experimental settings. Each query was also run five times for each XML document with a cold operating system cache. The time for query parsing and evaluation plan generation is excluded in all cases. The average running time was used in the performance evaluation.

8.1 Comparison study

We first compare the performance of MQF+ and SQF with the previous experimental results for MQF and XQuery reported in Sect. 5.3. The data sets we use are the same as those in our previous experiments: XMark (with a factor of 0.45) and DBLP. The time for building A–D index for XMark is 84 s, with a total size of 184 MB; the time for building A–D index for DBLP is 297 s, with a total size of 500 MB.

DBLP: Table 2 shows the actual execution time of schema-aware XQuery (XQuery) and mqf-embedded XQuery implemented using algorithm MQF, MQF+, and SQF, respectively, for the "XMP" queries on DBLP data. As can be seen, for all queries with selection conditions, both MQF+ and SQF outperform the basic stack-based algorithm MQF. The saving of computation cost can be mainly attributed to the reduced input size of the mqf operator by applying selection early. In particular, our A–D index-based algorithm MQF+ speeds up the processing of mqf-embedded XQuery from 22% (Q7) to as much as 33% (Q12), while the selectivity-based algorithm SQF speeds up query evaluation from 33% (Q12) to as much as 54% (Q7). In addition, the performance of SQF is consistently better than that of MQF+, because in SQF some input nodes can be pruned early without stack operations, while in MQF+ every input is involved in stack operations.

For queries without selection conditions (e.g., Q4), MQF and MQF+ are essentially the same and thus have the same performance. In such cases, we use the input size to decide the input order in SQF: for example, an input with a smaller size is considered to be more "selective" than that with a larger size. We expect that for queries with similar input size, the performance of SQF is no better than that of MQF or MQF+, as for such queries, almost all the inputs will appear in the final results, and thus the extra computation cost of checking A–D index surpasses any saving obtained from reducing inputs pushed onto the stack. But we expect that for queries with significant input size differences, where many inputs may be unqualified to be included the final results, SQF may still outperform both MQF and MQF+ by the early elimination of unqualified nodes without the need for stack operations. The results shown in Table 2 confirm our expectation: the mqf functions in those "XMP" queries without selections all have similar input size (e.g., Q3), and the performance of SQF is worse than that of MQF or MQF+, with an overhead of up to 18% (Q5).

Finally, for queries without MQF functions (Q8 and Q9), the performance is the same for XQuery and mqf-embedded XQuery, regardless of the implementations, as no computation of MQFs is needed.

XMark: Results for XMark are shown in Table 3. MQF+ accelerates the query evaluation for only one mqf-embedded query by a small fraction. For most XMark queries with selection conditions (e.g., Q1), the performance of MQF+ is actually worse than that of MQF. Given the performance improvement for "XMP" queries on DBLP by MQF+, this result may look surprising, but it is expected: MQF+ reduces computation cost by pushing selections before MQF operation, however, it also adds extra cost by checking A–D index (required for every input and its ancestors pushed onto the

Table 2 Performance (seconds) of XQuery, MQF, MQF+, and SQF for the "XMP" queries on DBLP

Category	Query	XQuery	MQF	MQF+	SQF
With selection	1	530	1533	1121	712
	7	552	1469	1140	672
	12	473	1088	732	730
Without selection	2	290	1173	1167	1233
	3	527	1421	1432	1546
	4	479	1665	1682	1884
	5	1132	2518	2496	2968
	6	371	1116	1120	1333
	10	367	1456	1478	1715
	11	511	1321	1316	1633
Without mqf function	8	240	243	238	241
	9	237	240	241	243

Table 3 Performance (seconds) of XQuery, MQF, MQF+, and SQF on XMark

Category	Query	XQuery	MQF	MQF+	SQF
With selection	1	8.5	28.9	39.6	24.1
	4	15.1	29.9	30.3	16.0
	5	6.6	7.5	12.5	7.4
	11	785	2387	2410	1101
	12	7.7	25.0	36	11.9
	14	11.5	27.9	25.8	19.4
	20	3.5	13.3	16	33
Without selection					
Similar input size	2	14.9	17.9	17.1	24.3
	10	93.2	110.3	115.5	116.7
	17	10.2	21.5	23.2	26.8
	19	19.9	33.2	33.5	33.9
Different input size	3	13.3	30.4	29.7	29.1
	8	25.6	78.1	77.6	57.4
	9	38.3	115.2	108.9	57.0
	13	3.0	3.6	3.5	3.8
	15	14.9	38.4	38.9	38.1
	16	15.5	54.8	56.5	45.7
Without mqf function	6	0.92	0.84	0.87	0.91
	7	4.74	4.76	4.69	4.75
	18	12.2	12.3	12.4	12.7

stack). Given a query with a single selective condition, but a overall large input size, such as the one in Q1, the extra overhead of checking A–D index (random disk access) rapidly exceeds the saving resulted from reduced size of one input. Consequently, MQF+ loses its advantage over MQF on such queries, while the "XMP" queries on DBLP benefited more from applying selection early, as they contain more than one selective conditions.

The performance of SQF suffers much less from the overhead of checking A–D index for stack nodes, as it is able to eliminate unqualified inputs early without pushing them onto the stack . In fact, SQF algorithm reduces the query evaluation cost for all the queries with selection conditions by up to 68% (Q12), with the exception of Q20. This discrepancy is due to the fact that Q20 contains multiple mqf functions, two of which contain no selections, but have inputs with large size. Consequently, the total overhead of checking A–D index exceeds the savings due to reduced input size.

For queries without selection condition (e.g., Q2), the results are similar to such queries on DBLP: MQF and MQF+ have the same performance, while SQF outperform MQF and MQF+ on queries with significant differences among input sizes (e.g., Q8), but usually underperform otherwise (e.g., Q2).

The performance of XQuery and mqf-embedded XQuery are also the same for queries without mqf functions (Q6 and Q7), since no computation of MQFs is needed.

8.2 Further examination

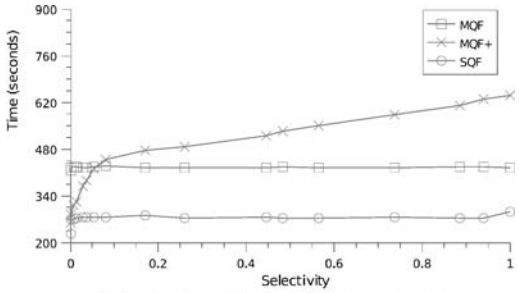
The comparison study in Sect. 8.1 demonstrates how MQF+ and SQF can help to reduce the evaluation cost of schema-free queries by supporting the integration of MQF operation into a query evaluation pipeline. The results also indicate that a number of factors may impact the performance of MQF+ and SQF, particularly the selectivity of selection conditions, and the size of inputs. We designed the following experiment to systematically investigate how the two factors affect the performance of different algorithms.

Method: Consider the following **mqf** function:

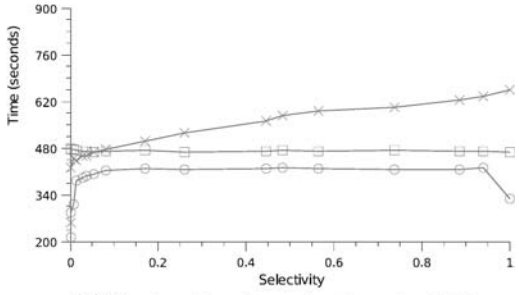
---

mqf($y, $t)
$y: inputs of type *year*.
$t: inputs of type *title*.

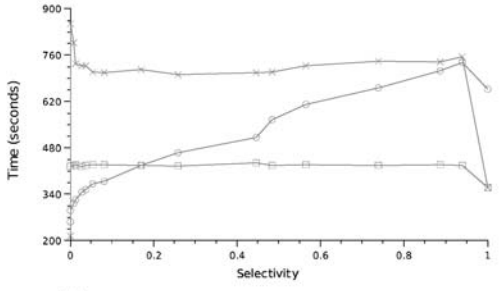
Different selection conditions can be applied to control the input size and selectivity of search conditions for the above mqf function. Specifically, we first varied the input size of nodes of type *title* by imposing different selection conditions on it. Then for each fixed input size of *title* nodes, we varied the selection conditions on nodes of type *year* to change its selectivity. We measured the actual execution time of evaluating the above mqf function on DBLP with different implementations: MQF, MQF+ and SQF. The results are shown in Fig. 19. The performance of algorithm MQF



(a) Fixed small input size $|n_{title}| = 5$



(b) Fixed medium input size $|n_{title}| = 1318$



(c) Fixed large input size $|n_{title}| = 418562$

Fig. 19 Performance of algorithm MQF, MQF+ and SQF, varying selectivity of *year* nodes and the input size of *title* nodes

can be viewed as a comparison base line, as it is consistent across different search conditions on nodes *year* given the same input size of nodes *title*, with only small variations among different input sizes of *title*, due to the different selection conditions applied after the MQF operator (e.g., selection condition applied on *title* in Fig. 19 is more expensive than the one used in Fig. 19).

Results Figure 19 displays the performance of different algorithms computing MQF under varying selectivities of selection conditions on *year*, when the input size of *title* node is very small ($|n_{title}| = 5$). As can been seen, SQF consistently outperforms MQF by approximately 36% regardless of the selectivity of search conditions on *year*. This advantage of SQF can be mainly attributed to its capability of eliminating unqualified nodes early without stack operations: the number of expensive stack operations is limited by the input with a smaller size. However, MQF+ performs better than MQF only when the search conditions on *year* is fairly

selective (selectivity < 0.1). When the search conditions become less selective, the performance of MQF+ quickly becomes worse than that of MQF, as the declining savings from reduced input size are surpassed by the increasing overhead incurred from checking A–D index for every stack node.

Given a relatively larger input size ($|n_{title}| = 1318$), the performance of both MQF+ and SQF decreases, as shown in Fig. 19. This observation can be attributed to the increased overhead from checking A–D index, with a overall larger input size. The performance of MQF+ suffers the most, as MQF+ is much more sensitive to the total number of input nodes than SQF. SQF still consistently outperforms MQF, but only by about 12% on average.

When the input size of *title* nodes is large (including all the *title* nodes in DBLP), as shown in Fig. 19, a significant performance decline can be observed for both MQF+ and SQF. The performance of MQF+ becomes consistently worse than that of MQF. Such degradation in performance is not surprising: given the overall significantly larger input size, the cost incurred in MQF+ from checking A–D index for every stack operation is overwhelmingly higher than the saving obtained from the reduced input size of MQF function by applying selection early. In particular, when the selectivity of search condition on *year* increases from 0 to 0.001, the evaluation cost of MQF function jumps from 212 to 852 s. This sharp rise of evaluation cost is also due to the overhead occurred from checking A–D index for every stack node. When the selectivity of search condition on *year* is zero, an empty result is returned without further computation, while even when the search condition on *year* is fairly selective, the computation of MQFs is still needed: given a large input size of *title* nodes, such overhead unavoidably becomes the dominant factor in determining the performance of MQF+. Meanwhile, SQF no longer consistently outperforms MQF. With a large input size of *title*, SQF can no longer benefit from first applying selection on *title*. Its performance now depends on the selectivity of *year*, winning over MQF only when this selectivity is less than 0.18. Given less selective conditions, the performance of SQF gradually degrades, and finally becomes the same as that of MQF. The benefit of reducing input size of MQF function and pruning unqualified nodes early without stack operations is gradually overtaken by the increasing cost of checking A–D index for base data information for the inputs.

Finally, the execution time of mqf function decreases when the selectivity of the search condition on *year* is increased to 1, regardless of the input size of *title*. This decline in query evaluation time is due to the absence of selection conditions.

Overall, we can draw the following conclusion on the performance of three different algorithms introduced in this paper: when there exist selective conditions, or significant differences among input sizes, SQF tends to perform better than MQF or MQF+; otherwise, MQF is likely to be a better choice.

9 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 [5], DBXplorer [6], DISCOVER [26], and [21] attempt to use keyword searches in relational database. In those studies, a database is viewed as a graph with objects as nodes and relationships between objects as edges; subgraphs of the database are returned as answers to the original keyword query. A similar approach has also been taken to employ keyword search in XML documents (e.g., XKeyword [25] and XRANK [22]). 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 names from textual content; (2) they cannot express complex query semantics.

Several attempts have also been made to support information retrieval style search by expanding XQuery or other structured query languages (e.g., [9, 10, 13, 16, 19, 20, 35]). In particular, XXL [35], XIRQL [20], ELIXIR [16], and JuruXML [13] focus on vague matching for limited XPath predicates, while FleXpath [9] proposes a more comprehensive formalization for structural relaxation in XML queries. Different relaxation and ranking techniques have been proposed in these works to support vague matching for XPath predicates. These approaches require a user to understand XPath, and have some knowledge on the document structure to specify meaningful XPath queries as the base for query relaxation and result ranking. In cases where a user is unaware of the document structure, and cannot specify meaningful XPath predicates, they do not exploit any document structure. Unlike the above fuzzy matching methods, Schema-Free XQuery provides well defined exact matches, and thus requires no scoring or ranking. Other approaches (e.g., LOREL [30] and Meet [34]) created query languages to enable keyword search in XML documents and exploit some structural information that is not specified in the query. The major differences between those approaches and ours are that we eliminate any requirement for path expressions. We also exploit the document structure better to identify results that are more meaningful.

A recent closely related work is XSEarch [17], 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 of expressing complex knowledge semantics. The MQF operator, on the other hand, takes full advantage of well-defined XQuery, and enables the user to take more control of search results without knowing document structure.

In addition, query mediation systems such as the REVERE system [23] allow query answering across schemas by deploying schema mapping and query rewriting techniques. Users are still required to have extensive knowledge of at least one schema to pose queries.

Term expansion has been studied extensively in the information retrieval literature [3, 12, 18, 31]. Various techniques have been also proposed in previous work on fuzzy matching for XML queries [20, 33] to address term expansion. In fact, recently released W3C Working Draft on XQuery Full-Text [39] already includes a framework to support the integration of (ontology-based) term expansion. Compared with techniques such as the one proposed in XIRQL [20], where complex DTD extension is required, our approach to term expansion is straightforward: we utilize WordNet [3] for synonym expansion, and depend on domain-specific ontology for hypernym and hyponym expansion. Indices are designed to support efficient expansion. One may view our method as a specific implementation that fits into the framework of XQuery Full-Text. Note, however, we only deal with the term expansion for tag names in our work. This special handling of tag name is due to the fact that tag name confusion tends to have greater impact on search results, especially in Schema-Free XQuery, as it directly affects the MQFs generated as query context. It would be interesting to investigate how more complex term expansion methods, such as the distance metric in [31], can be integrated into our work, but it is out of the scope of this paper.

Finally, much work has been done on efficiently computing LCAs [24, 32, 40, 41]. In [24, 32, 40], constant time algorithms for main memory data structures obtained by preprocessing of the tree are proposed. Those works do not consider disk access minimization when the structures cannot fit into memory, while in both our work and [41], disk access is minimized such that no disk access is necessary when computing the LCAs of given nodes. However, the methods proposed in [41] are based on some special properties of *Smallest LCA*. They cannot be easily adopted to compute MQFs, as MQF is much more complex than Smallest LCA, and does not exhibit the same properties.

10 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." By bridging the gap between the limited user knowledge of an XML document and the actual document schema, Schema-Free XQuery also exhibits great potential for supporting more flexible yet powerful user interface for querying XML documents, as demonstrated by our recent work NaLIX [29], a Natural Language Interface for Querying XML.

We also devised a stack-based algorithms for the MQF computation at the heart of schema-free query. Experiments show that this algorithm is up to 16 times faster than a basic MQF computation using standard operators. Schema-Free XQuery evaluated with this stack-based algorithm incurs an overhead no more than three times the execution time of an equivalent schema-aware query. To further improve the query processing of schema-free queries by integrating MQF calculation in the query evaluation pipeline, we designed an index structure called Ancestor–Descendant Summarization (A–D) index to enable fast access to base data. We also developed two different algorithms utilizing A–D index, namely MQF+ and SQF, to further reduce the evaluation cost of schema-free query evaluation.

Future directions for research include investigating techniques for applying MQF to queries involving references. We also intend to use more sophisticated IR techniques where appropriate in schema-free queries.

Acknowledgements This work was supported in part by the United States National Science Foundation (NSF) under grants NSF IIS-0438909, IIS-0219513, by NIH under grant number LM08106-01, and by a gift from Microsoft. We would like to thank the anonymous reviewers for their constructive suggestions. We also would like to thank Adriane Chapman for her help with proofreading this paper.

References

1. INEX: http://inex.is.informatik.uni-duisburg.de:2004/
2. TIMBER: http://www.eecs.umich.edu/db/timber
3. WordNet: http://www.cogsci.princeton.edu/~wn/
4. XMark: http://monetdb.cwi.nl/xml/index.html
5. Aditya, B. et al.: BANKS: Browsing and keyword searching in relational databases. VLDB (2002)
6. Agrawal, S. et al.: DBXplorer: a system for keyword-based search over relational databases. ICDE (2002)
7. Al-Khalifa, S. et al.: Structural joins: A primitive for efficient XML query pattern matching. ICDE (2001)
8. Al-Khalifa, S. et al.: Querying structured text in an XML database. SIGMOD (2003)
9. Amer-Yahai, S. et al.: FleXPath: Flexible structure and full-text querying for XML. SIGMOD (2004)
10. Amer-Yahia, S. et al.: TeXQuery: A full-text search extension to XQuery. WWW (2004)
11. Bruno, N. et al.: Holistic twig joins: Optimal XML pattern matching. SIGMOD (2002)
12. Burton-Jones, A. et al.: A heuristic-based methodology for semantic augmentation of user queries on the Web. ER (2003)
13. Carmel, D. et al.: Searching XML documents via XML fragments. SIGIR (2003)

14. Chamberlin, D.: XQuery: An XML query language. IBM Syst. J. **41**, 597–615 (2003)
15. Chien, S.-Y. et al.: Efficient structural joins on indexed XML documents. VLDB (2002)
16. Chinenyanga, T.T., Kushmerick, N.: Expressive and efficient ranked querying of XML data. WebDB (2001)
17. Cohen, S. et al.: XSEarch: A semantic search engine for XML. VLDB (2003)
18. Deerwester, S. et al.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. (1990)
19. Florescu, D. et al.: Integrating keyword search into XML query processing. Comput. Netw. **33**, 119–135 (2000)
20. Fuhr, N., Großjohann, K.: XIRQL: An extension of XQL for information retrieval. SIGIR (2000)
21. Goldman, R. et al.: Proximity search in databases. VLDB (1998)
22. Guo, L. et al.: XRANK: Ranked keyword search over XML documents. SIGMOD (2003)
23. Halevy, A. et al.: Crossing the structure chasm. CIDR (2003)
24. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
25. Hristidis, V. et al.: Keyword proximity search on XML graphs. ICDE (2003)
26. Hristidis, V., Papakonstantinou, Y.: Discover: Keyword search in relational databases. VLDB (2002)
27. Jagadish, H.V. et al.: TIMBER: A native XML database. VLDB J. **11**(4), 274–291 (2002)
28. Ley, M.: DBLP bibliography (2003)
29. Li, Y. et al.: NaLIX: An interactive natural language interface for querying XML. SIGMOD (2005)
30. Quass, D. et al.: Querying semistructured heterogeneous information. DOOD (1995)
31. Resnik, P.S.: Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural langauge. J. Artif. Intell. Res. **11**, 95–130 (1999)
32. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. **17**(6), 1253–1262 (1988)
33. Schlieder, T.: Similarity search in XML data using cost-based query tranformations. SIGMOD (2001)
34. Schmidt, A. et al.: Querying XML documents made easy: Nearest concept queries. ICDE (2001)
35. Theobald, A., Weikum, G.: The index-based XXL search engine for querying XML data with relevance ranking. EDBT (2002)
36. W3C: XML Query Use Cases. W3C Working Draft. Available at http://www.w3.org/TR/xquery-use-cases/ (2003)
37. W3C: XML Schema. W3C Recommendation. Available at http://www.w3.org/XML/Schema (2003)
38. W3C: XQuery 1.0. W3C Working Draft. Available at http://www.w3.org/TR/xquery/ (2004)
39. W3C: XQuery 1.0 and XPath 2.0 Full-Text. W3C Working Draft. Available at http://www.w3.org/TR/xquery-full-text/ (2005)
40. Wen, Z.: New algorithms for the LCA problem and the binary tree reconstruction problem. Inf. Process. **51**(1), 11–16 (1994)
41. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. SIGMOD (2005)