The VLDB Journal (2009) 18:543–570
DOI 10.1007/s00778-008-0126-x

SPECIAL ISSUE PAPER

Efficient keyword search over virtual XML views

Feng Shao · Lin Guo · Chavdar Botev · Anand Bhaskar · Muthiah Chettiar · Fan Yang · Jayavel Shanmugasundaram

Received: 23 February 2008 / Revised: 13 September 2008 / Accepted: 27 November 2008 / Published online: 23 January 2009

Abstract Emerging applications such as personalized portals, enterprise search, and web integration systems often require keyword search over semi-structured views. However, traditional information retrieval techniques are likely to be expensive in this context because they rely on the assumption that the set of documents being searched is materialized. In this paper, we present a system architecture and algorithm that can efficiently evaluate keyword search queries over *virtual* (unmaterialized) XML views. An interesting aspect of our approach is that it exploits indices present on the base data and thereby avoids materializing large parts of the view that are not relevant to the query results. Another feature of the algorithm is that by solely using indices, we can still score the results of queries over the virtual view, and the resulting scores are the same *as if* the view was materialized. Our performance evaluation using the INEX data set in the Quark (Bhaskar et al. in Quark: an efficient XQuery full-text implementation. In: SIGMOD, 2006) open-source XML database system indicates that the proposed approach is scalable and efficient.

Keywords Keyword search · XML views · Document projections · Document pruning · Top-K

F. Shao (✉) · L. Guo · C. Botev · A. Bhaskar · M. Chettiar · F. Yang
Cornell University, Ithaca, NY, 14853, USA
e-mail: fshao@cs.cornell.edu

L. Guo
e-mail: guolin@cs.cornell.edu

C. Botev
e-mail: cbotev@cs.cornell.edu

A. Bhaskar
e-mail: ab394@cornell.edu

M. Chettiar
e-mail: mm376@cornell.edu

F. Yang
e-mail: yangf@cs.cornell.edu

J. Shanmugasundaram
Yahoo! Research, Santa Clara, CA 95054, USA
e-mail: jaishan@yahoo-inc.com

1 Introduction

Traditional information retrieval systems rely heavily on a fundamental assumption that the set of documents being searched is materialized. For instance, the popular inverted list organization and associated query evaluation algorithms [5,35] assume that the (materialized) documents can be parsed, tokenized, and indexed when the documents are loaded into the system. Further, techniques for scoring results such as TF-IDF [35] rely on statistics gathered from materialized documents such as term frequencies (number of occurrences of a keyword in a document) and inverse document frequencies (the inverse of the number of documents that contain a query keyword). Finally, even document filtering systems, which match streaming documents against a set of user keyword search queries (e.g., [10,17]), assume that the document is fully materialized at the time it is handed to the streaming engine, and all processing is tailored for this scenario.

In this paper, we argue that there is a rich class of semi-structured search applications for which it is undesirable or impractical to materialize documents. We illustrate this claim using two examples.

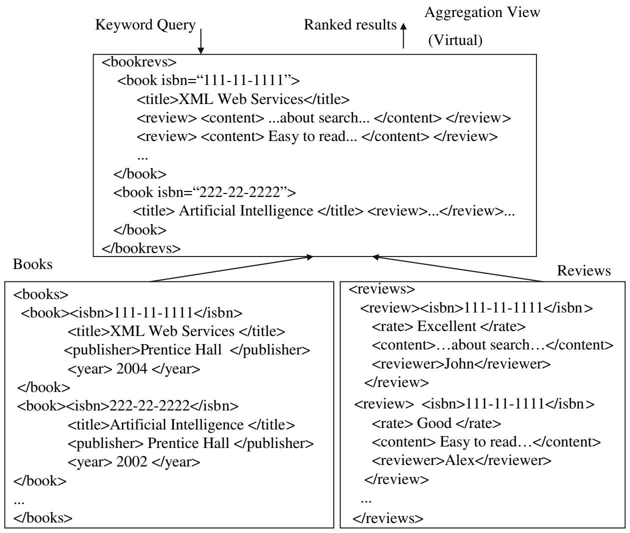
Personalized Views. Consider a large online web portal such as MyYahoo[1] that caters to millions of users. Since

[1] http://my.yahoo.com.

different users may have different interests, the portal may wish to provide a personalized view of the content to its users (such as books on topics of interest to the user along with their reviews, and latest headlines along with previous related content seen by the user, etc.), and allow users to search such views. As another example, consider an enterprise search platform such as Microsoft Sharepoint[2] that is available to all employees. Since different employees may have different permission levels, the enterprise must provide personalized views according to specific levels, and allow employees to search only such views. In such cases, it may not be feasible to materialize all user views because there are many users and their content is often overlapping, which could lead to data duplication and its associated space-overhead. In contrast, a more scalable strategy is to define virtual views for different users of the system, and allow users to search over their virtual views.

Information Integration. Consider an information integration application involving two query-able XML web services: the first service provides books and the second service provides reviews for books. Using these services, an aggregator wishes to create a portal in which each book contains its reviews nested under it. A natural way to specify this aggregation is as an XML view, which can be created by joining books and reviews on the isbn of the book, and then nesting the reviews under the book (Fig. 1). Note that the view is often virtual (unmaterialized) for various reasons: (a) the aggregator may not have the resources to materialize all the data, (b) if the view is materialized, the contents of the view may be out-of-date with respect to the base data, or maintaining the view in the face of updates may be expensive, and/or (c) the data sources may not wish to provide the entire data set to the aggregator, but may only provide a sub-set of the data in response to a query. While current systems (e.g., [9,15,20]) allow users to query virtual views using query languages such as XQuery, they do not support ranked keyword search queries over such views.

The above applications raise an interesting challenge: how do we efficiently evaluate keyword search queries over virtual XML views? One simple approach is to materialize the entire view at query evaluation time and then evaluate the keyword search query over the materialized view. However, this approach has obvious disadvantages. First, the cost of materializing the entire view at runtime can be prohibitive, especially since only a few documents in the view may contain the query keywords. Further, users issuing keyword search queries are typically interested in only the results with highest scores, and materializing the entire view to produce only top few results is likely to be expensive.



```
<bookrevs>
   <book isbn="111-11-1111">
      <title>XML Web Services</title>
      <review> <content> ...about search... </content> </review>
      <review> <content> Easy to read... </content> </review>
      ...
  </book>
  <book isbn="222-22-2222">
     <title> Artificial Intelligence </title> <review>...</review>...
  </book>
</bookrevs>
```

```
<books>
 <book><isbn>111-11-1111</isbn>
         <title>XML Web Services </title>
        <publisher>Prentice Hall  </publisher>
         <year> 2004 </year>
 </book>
 <book><isbn>222-22-2222</isbn>
         <title>Artificial Intelligence </title>
         <publisher> Prentice Hall </publisher>
         <year> 2002 </year>
</book>
...
</books>
```

```
<reviews>
  <review><isbn>111-11-1111</isbn>
    <rate> Excellent </rate>
    <content>…about search…</content>
    <reviewer>John</reviewer>
   </review>
 <review>  <isbn>111-11-1111</isbn>
    <rate> Good </rate>
    <content> Easy to read…</content>
    <reviewer>Alex</reviewer>
   </review>
  ...
 </reviews>
```

Fig. 1 An XML view associating books and reviews

To address the above issues, we propose an alternative strategy for efficiently evaluating keyword search queries over virtual XML views. The key idea is to use regular indices, including inverted list and XML path indices that are present on the base data to efficiently evaluate keyword search over views. The indices are used to efficiently identify the portion of the base data that is relevant to the current keyword search query so that only the top ranked results of the view are actually materialized and presented to the user.

The above strategy poses two main challenges. First, XML view definitions can be fairly complex, involving joins and nesting, which leads to various subtleties. As an illustration, consider Fig. 1. If we wish to find all books with nested reviews that contain the keywords "XML" and "search", then ideally we want to materialize only those books and reviews such that *together* contain the keywords "XML" and "search" (even though no book or review may *individually* contain both the keywords). However, we cannot determine which reviews belong to which book (to check whether they together contain both the keywords) without actually joining the books and reviews on the isbn, which is a data value. This presents an interesting dilemma: how do we selectively extract some fields needed for determining related items in the view (e.g., isbn) without actually materializing the entire view?

The second challenge stems from ranking the keyword search results. As mentioned earlier, popular ranking methods such as TF-IDF require statistics gathered from the documents being searched. How do we efficiently compute statistics on the view from the statistics on the base data, so that the resulting scores and rank order of the query results is exactly the same as when the view is materialized?

Our solution to the above problem is a three-phase algorithm that works as follows. In the first phase, the algorithm analyzes the view definition and query keywords to identify

[2] http://www.microsoft.com/sharepoint.

a *query pattern tree* (or QPT) for each data source (such as books and reviews); the QPT represents the precise parts of the base data that are required to compute the potential results of the keyword search query. In the second phase, the algorithm uses existing inverted and path indices on the base data to compute *pruned document trees* (or PDT) for each data source; each PDT contains only small parts of the base data tree that correspond to the QPT. The PDT is constructed *solely* using indices, without having to access the base data. In this phase, the algorithm also propagates keyword statistics in the PDTs. In the third phase, the query is evaluated over the PDTs, and the top few results are expanded into the complete trees; this is the only phase where the base data is accessed (for the top few results only).

We have experimentally compared our approach with two alternatives: the naive approach that materializes the entire view at query time, and TLC/GTP [13,33] with TermJoin [2], which is a state-of-the-art implementation of integrating structure and keyword search queries. Our experimental results show that our approach is *more than ten times faster* than these alternatives, due to the following two reasons: (1) we use path indices to efficiently create PDTs, thereby avoiding more expensive structural joins, and (2) we selectively materialize the element values required during query evaluation using indices, without having to access the base data. We have also compared our PDT generation with the technique for projecting XML documents [28]; again our approach is more than an order of magnitude faster because we generate PDTs solely using indices.

In summary, we believe that the proposed approach is the first optimized end-to-end solution for efficient keyword search over virtual XML views. The specific contributions of this paper are

– A system architecture for efficiently evaluating keyword search queries over virtual XML views (Sect. 3).
– Efficient algorithms for generating pruned XML elements needed for query evaluation and scoring, by solely using indices (Sect. 4).
– Generalizations and optimizations to the proposed algorithms to handle a larger class of complex XML views that contain descendant axes, element wild cards and repeating element tags (Sect. 5).
– Formalization of the invariants of the generalized algorithms along with their proof of correctness (Sect. 6 and Appendix A).
– Evaluation and comparison of the proposed approach using the 500MB INEX dataset[3] (Sect. 7).

[3] http://inex.is.informatik.uni-duisburg.de:2004.

We note that Sects. 5, 6, and Appendix A are new sections that significantly generalize the earlier results that appeared in [39].

There are some interesting optimizations and extensions to the proposed approach that are not explored in this paper. First, the proposed approach produces *all* pruned view elements, so that each element is scored and only the top few results are fully materialized. While this deferred materialization already leads to significant performance gains, an even more efficient strategy might be to avoid producing the pruned view elements that do not make it to the top few results. This problem, however, turns out to be non-trivial because of the presence of non-monotonic operators such as group-by that are common in XML views (please see the conclusion for more details). Second, the current focus of this paper is on aspects related to system efficiency; consequently, the discussion on scoring is limited to simple XML scoring methods based on TF-IDF [35]. Generalizing the proposed approach to deal with more sophisticated XML scoring functions (e.g., [3,22,29]) is another interesting direction for future work.

2 Background and problem definition

We first describe some background on XML, before presenting our problem definition.

2.1 XML Documents and queries

An XML document consists of nested XML elements starting with the root element. We support complex types with mixed content, hence each element can have values, attributes, and nested subelements. Figure 1 shows an example XML document representing books with nested reviews. Each $\langle book \rangle$ element has $\langle title \rangle$ and $\langle review \rangle$ subelements nested under it. The $\langle book \rangle$ element also has the isbn attribute. For ease of exposition, we treat attributes as though they are sub-elements. While XML elements can also have references to other elements (IDREFs), they are treated and queried as values in XML; hence we do not model this relationship explicitly for the purposes of this paper. In order to capture the text content of elements, we use the predicate $contains(u, k)$, which returns true iff the element u directly or indirectly contains the keyword k (note that k can appear in the tag name or text content of u or its descendants).

An XML database instance D can be modeled as a set of XML documents. An XML query Q can be viewed as a mapping from a database instance D to a sequence of XML documents/elements (which represents the output of the query). More formally, if UD is the universe of XML database instances and S is the universe of sequences of XML documents/elements, then $Q : UD \rightarrow S$. Thus, we

```
let $view :=
for $book in fn:doc(books.xml)/books//book
where $book/year > 1995
return <bookrevs>
           <book> {$book/title} </book>,
           {for $rev in fn:doc(reviews.xml)/reviews//review
            where $rev/isbn = $book/isbn
            return $rev/content}
       </bookrevs>

for $bookrev in $view
where $bookrev ftcontains('XML' & 'Search')
return $bookrev
```

Fig. 2 Keyword search over XML view

use the notation $Q(D)$ to denote the result of evaluating the query Q over the database instance D. A query Q is typically specified using an XML query language such as XQuery. An XML view is simply represented as an XML Query. For instance, the variable $\$view$ in Fig. 2 corresponds to an XQuery query/view which nests *review* elements in the review document under the corresponding *book* element in the book document. We thus use the term view and query interchangeably for the rest of the paper. Further, we use the following notation for reasoning about sequences of elements. Given a sequence of elements s, $e \in s$ is true iff the element e is present in the sequence s.

2.2 XML scoring

An important issue for keyword search queries is scoring the results. There have been many proposals for scoring XML keyword search results [3,4,21,22,29]. As mentioned in Sect. 1, in the paper we focus on the commonly used TF-IDF method proposed in the context of XML documents [21]. In this context, tf and idf values are calculated with respect to XML *elements*, instead of entire documents as in the traditional information retrieval. Specifically, given an XML view V over a database D, the TF-IDF method defines two measures:

- $tf(e, k)$, which is the number of distinct occurrences of the keyword k in element e and its descendants (where $e \in V(D)$), and
- $idf(k) = \frac{|V(D)|}{|\{e|e \in V(D) \wedge contains(e,k)\}|}$ (the ratio of the number of elements in the view result V(D) to the number of elements in V(D) that contain the keyword k).

Given the above measure, the score of a result element e for a keyword search query Q is defined to be $score(e, Q) = \Sigma_{k \in Q}(tf(e, k) \times log(idf(k)))$. The score can be further normalized using various methods proposed in the literature [43].

2.3 Problem definition

We use a set of keywords $Q = \{k_1, k_2, \ldots, k_n\}$ to represent a keyword search query and we support both conjunctive and disjunctive queries.

First, we define the problem of conjunctive keyword search over views as follows.

Problem CONJ-KS. Given a view V defined over a database D, the result of a keyword search query Q, denoted as *CONJ-RES(Q,V,D)*, is the sequence s such that

- $\forall e\ e \in s \Rightarrow e \in V(D)$, and
- $\forall e\ e \in s \Rightarrow \forall q \in Q\ contains(e, q)$, and
- $\forall e\ (e \in V(D) \wedge \forall q \in Q contains(e, q)) \Rightarrow e \in s$

Figure 2 illustrates a keyword query *{'XML', 'Search'}* over the view corresponding to the variable $\$view$. Given the definition of score in the previous section, we can further define the problem of *ranked* keyword search as follows.

Problem Ranked-CONJ-KS. Given a view V defined over a database D and the number of desired results K, the result of a ranked keyword query Q is the ordered list of K elements with highest scores in *CONJ-RES(Q,V,D)* in decreasing order of score, where we break ties arbitrarily.

The disjunctive queries can be defined similarly as follows.

Problem DISJ-KS. Given a view V defined over a database D, the result of a keyword search query Q, denoted as *CONJ-RES(Q,V,D)*, is the sequence s such that

- $\forall e\ e \in s \Rightarrow e \in V(D)$, and
- $\forall e\ e \in s \Rightarrow \exists q \in Q\ contains(e, q)$, and
- $\forall e\ (e \in V(D) \wedge \exists q \in Q contains(e, q)) \Rightarrow e \in s$

Problem Ranked-DISJ-KS. Given a view V defined over a database D and the number of desired results K, the result of a ranked keyword query Q is the ordered list of K elements with highest scores in *DISJ-RES(Q,V,D)* in decreasing order of score, where we break ties arbitrarily.

Our running example is an example of the problem Ranked-CONJ-KS. While the rest of the paper is mostly based on the running example, the ideas can easily apply on Ranked-DISJ-KS. In fact, the difference between Ranked-CONJ-KS and Ranked-DISJ-KS only influences the last expansion phase. Our PDT algorithms, which are the main contributions of the paper, remain the same.

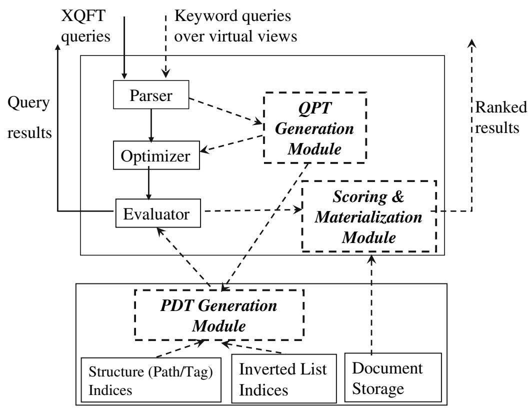


Fig. 3 Keyword query processing architecture

3 System overview

3.1 System architecture

Figure 3 shows our proposed system architecture and how it relates to traditional XML full-text query processing. The top big box denotes the query engine sub-system and the bottom big box denotes the storage and index subsystem. The solid lines show the traditional query evaluation path for full-text queries (e.g., [6,16,26,31]). The query is parsed, optimized and evaluated using a mix of structure and inverted list indices and document storage. However, as mentioned in the introduction, traditional query engines are not designed to support efficient keyword search queries over views. Consequently, they either disallow such queries (e.g., [16,31]), materialize the entire view before evaluating the keyword search query (e.g., [6]), or do not support such queries efficiently (e.g., [26]), as verified in our performance study (Sect. 7).

To efficiently process keyword search queries over views, we adapt the existing query engine architecture by adding three new modules (depicted by dashed boxes in Fig. 3). The modified query execution path (depicted by dashed lines in Fig. 3) is as follows. On detecting a keyword search query over a view that satisfies certain conditions (clarified at the end of this section), the parser redirects the query to the Query Pattern Tree (QPT) Generation Module. The QPT, which is a generalization of the Generalized Tree Patterns [13], identifies the precise parts of the base data that are required to compute the results of the keyword search query. The QPT is then sent to the Pruned Document Tree (PDT) Generation Module. This module generates PDTs (i.e., a projection of the base data that conforms to the QPT) using *only* the path indices and inverted list indices; consequently, the generation of PDTs is expected to be fast and cheap.

The QPT Generation Module also rewrites the original query to go over PDTs instead of the base data and sends it to the *traditional* query optimizer and evaluator. Note that our proposed architecture requires *no changes* to the XML query evaluator, which is usually a large and complex piece of code. The rewritten query is then evaluated using PDTs to produce the view that contains all view elements with pruned content (determined using path indices), along with information about scores and query keywords contained (determined using inverted indices). These elements are then scored by the Scoring & Materialization Module, and only those with highest scores are fully materialized using document storage.

Our current implementation supports views specified using a powerful subset of XQuery, including XPath expressions with named child and descendant axes, predicates on leaf values, nested FLWOR expressions and non-recursive functions. We currently do not support predicates on the string values of non-leaf elements and other XPath axes such as sibling and position based predicates, although it is possible to extend our system to handle these axes by using an underlying structure index that supports these axes (e.g., [14]). The current implementation only supports conjunctive and disjunctive containment predicates and it is possible to extend the architecture to support more complex full-text predicates such as distances between different keywords.

The supported view grammar is given below, where `Expr` is the root production and `VAR` and `TAGNAME` correspond to variables and element tag names, respectively.

```
Expr :- PathExpr | FLWORExpr | CondExpr
     | FunctionCall | FunctionDecl
PathExpr :- 'fn:doc(' QName ')' | VAR | '.'
          | ( 'fn:doc(' QName ')' | VAR | '.' )
            ('/'|'//') PathTailExpr
          | PathExpr '[' PredExpr ']'
PathTailExpr :- TAGNAME | TAGNAME ('/'|'//') PathTailExpr
PredExpr :- PathExpr | PathExpr Comp Literal
            | PathExpr Comp PathExpr
Comp :- '=' | '<' | '>'
CondExpr :- 'if' Expr 'then' Expr 'else' Expr
FLWORExpr :- (ForClause | LetClause)+
             (WhereClause)? ReturnClause
ForClause :- 'for' VAR 'in' PathExpr
LetClause :- 'let' VAR ':=' PathExpr
WhereClause :- 'where' PredExpr
ReturnClause :- 'return' RetExpr
RetExpr :- Expr
      | '<' TAGNAME '>' ('{' RetExpr '}')* '</' TAGNAME '>'
      | Expr ',' Expr
FunctionCall :- QName '(' (PathExpr (',' PathExpr)*)? ')'
FunctionDecl :- 'declare' 'function' QName
             '(' ParamList? ')' ? '{' Expr '}'
ParamList :- VAR (',' VAR)*
```

3.2 XML storage and indexing

Since our system architecture exploits indices on the base data to generate PDTs, we now provide some necessary background on XML storage and indexing techniques.

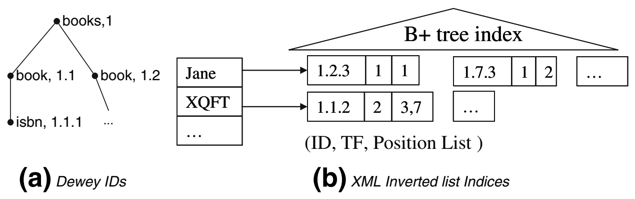


Fig. 4 Illustrating XML storage and indices

B+-Tree

Path	Value	IDList
…	…	…
/books/book/isbn	"111-111-1111"	1.1.1,1.3.1
/books/book/isbn	"222-222-2222"	1.2.1
…	…	…
/books/book/author/fn	"Jane"	1.2.4.3, 1.7.4.3

Path-Values Table

Fig. 5 XML path indices

One of the key concepts in XML storage is the notion of element ids, which is a way to uniquely identify an XML element. One popular id format is Dewey IDs which has been shown to be effective for search [22] and update [32] queries. Dewey IDs is a hierarchical numbering scheme where the ID of an element contains the ID of its parent element as a prefix. An example XML document in which Dewey IDs are assigned to each node is shown in Fig. 4a.

Another important aspect is XML indexing. At a high-level, there are two types of XML indices: path indices and inverted list indices (these indices can sometimes be combined [27]). Path indices are used to evaluate XML path and twig (i.e., branching path) queries. Inverted list indices are used to evaluate keyword search queries over (materialized) XML documents. We now describe representative implementations for each type of index.

One effective way to implement path indices is to store XML paths with values in a relational table and use indices such as B+-tree [12,41] for efficient probes. Figure 5 shows an example path index. As shown, the *Path-Values* index table contains one row for each unique (Path, Value) pair, where path represents a path from the root to an element in the document, and value represents the atomic value of the last element on the path. For each unique (Path, Value) pair, the table stores an *ID List*, which is the list of ids of all elements on the path corresponding to Path with that atomic *value* (paths without corresponding values are associated with a null value). A B+-tree index is built on the (Path, Value) pair. Queries are evaluated as follows. First, a path query with value predicates such as /book/author/fn[. = 'Jane'] is evaluated by probing the index using the search key (Path, 'Jane'). Second, a path query without value predicates is evaluated by merging lists of IDs corresponding to the path, which are retrieved using Path, the prefix of the composite key. For path queries with descendant axes, such as /book//fn, the index is probed for each full data path (e.g., /book/name/fn), and the lists of result ids are merged. Finally, twig queries are evaluated by first evaluating each individual path query and then merging the results based on the Dewey ID.

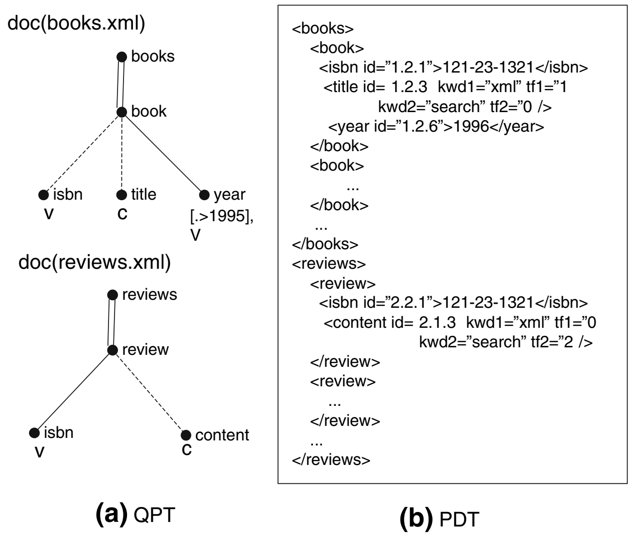
The second type of XML indices are inverted list indices. XML inverted list indices (e.g., [22,30,42]) typically store for each keyword in the document collection, the list of XML elements that *directly* contain the keyword. Figure 4 shows an example inverted list for our example document. In addition, an index with Dewey IDs as the keys, such as a B+-tree, is usually built on top of each inverted list so that we can efficiently check whether a given element contains a keyword.

3.3 QPT generation module

The QPT Generation Module (Fig. 3) generates QPTs from an XML view. We illustrate the QPT using the view shown in Fig. 2. In order to *evaluate* this view query, we only need a small subset of the data, such as the isbns of books and isbns of reviews (which are required to perform a join). It is only when we want to *materialize* the view results do we need additional content such as the titles of books and content of reviews. The QPT is essentially a principled way of capturing this information.

The QPT is a generalization of the Generalized Tree Patterns (GTP) [13], which was originally proposed in the context of evaluating complex XQuery queries. The GTP captures the structural parts of an XML document that are required for query processing. We refer the reader to [13] for the formal definition of GTP. The QPT augments the GTP structure with two annotations, one that specifies which parts of the structure and associated data values are required during query evaluation, and the other that specifies which parts are required during result materialization.

Figure 6a shows the QPTs for the book and review documents referenced in our running example. We first describe features present in the GTP. First, each QPT is associated with an XML document (determined by the view query). Second, as is usual in twigs, a double line edge denotes ancestor/descendant relationship and a single line edge denotes a parent/child relationship. Third, nodes are associated with tag names and (possibly) predicates. For instance, the *year* node in Fig. 6a is associated with a predicate $>$ 1995. Finally, edges in the QPT are either optional (represented by dotted lines) or mandatory (represented by solid lines). For example, in Fig. 6a, the edge between *book* and *isbn* is optional, because a book can be present in the view result even if it does not have an isbn; the edge between *review* and *isbn* is mandatory, because a review is of no relevance to query execution unless it has an isbn (otherwise, it does not join



```
<books>
  <book>
   <isbn id="1.2.1">121-23-1321</isbn>
    <title id= 1.2.3  kwd1="xml" tf1="1
              kwd2="search" tf2="0 />
     <year id="1.2.6">1996</year>
  </book>
  <book>
        ...
  </book>
   ...
</books>
<reviews>
  <review>
   <isbn id="2.2.1">121-23-1321</isbn>
    <content id= 2.1.3  kwd1="xml" tf1="0
                     kwd2="search" tf2="2 />
  </review>
  <review>
     ...
  </review>
  ...
</reviews>
```

Fig. 6 QPTs and PDTs of *book* and *review*

with any book and is hence irrelevant to the content of the view).

The new features in the QPT are node annotations ‘c’ and ‘v’, where ‘c’ indicates that the content of the node is propagated to the view output, and ‘v’ indicates that the value of the node is required to evaluate the view. In our example, the ‘isbn’ nodes in both the book and review QPT are marked with a ‘v’ since their values are required for performing a join operation; the ‘title’ and ‘content’ nodes are marked as ‘c’ nodes since their content is propagated to the view output, and is required *only* during materialization. Note that a node can be marked with both a ‘v’ and a ‘c’ if it is used during evaluation and propagated to the view output, although there is no instance of this case in our example.

We now introduce some notation that is used in subsequent sections. A QPT is a tree Q = (N, E) where N is the set of nodes and E is the set of edges. For each node n in N, n.tag is its tag name, n.preds is the set of predicates associated with n, and n.ann is its node annotation(s), which can be ‘v’, ‘c’, both, or neither. For each edge e in E, e.parent and e.child are the parent and child node of e, respectively; e.axis is either ‘/’ or ‘//’ corresponding to an XPath axis, and e.ann is either ‘o’ or ‘m’ corresponding to an optional or a mandatory edge.

4 PDT generation module

We now turn our attention to the PDT Generation Module (Fig. 3), which is one of the main technical contributions in the paper. The PDT Generation Module efficiently generates a PDT for each QPT. Intuitively, the PDT only contains elements that correspond to nodes in the QPT and only contains element values that are required during query evaluation. For example, Fig. 6b shows the PDT of the *book* document for its QPT shown in Fig. 6a. The PDT only contains elements corresponding to the nodes $books$, $book$, $isbn$, $title$, and $year$, and only the elements $isbn$ and $year$ have values.

Using PDTs in our architecture offers two main advantages. First, the query evaluation is likely to be more efficient and scalable because the query evaluator processes pruned documents which are much smaller than the underlying data. Further, using PDTs allows us to use the regular (unmodified) query evaluator for keyword query processing.

We note that the idea of creating small documents is similar to projecting XML documents (PROJ for short) proposed in [28]. There are, however, several key differences, both in semantics and in performance. First, while PROJ deals with isolated paths, we consider twigs with more complex semantics. As an example, consider the QPT for the $book$ document in Fig. 6a. For the path $/books//book/isbn$, PROJ would produce and materialize all elements corresponding to $book$ (and its subelements corresponding to $isbn$). In contrast, we only produce $book$ elements which have $year$ subelements whose values are greater than 1995, which is enforced by the entire twig pattern. Second, instead of materializing every element as in PROJ, we selectively materialize a (small) portion of the elements. In our example, only the elements corresponding to $isbn$ and $year$ are materialized. Finally, the most important difference is that we construct the PDTs by solely using indices, while PROJ requires full scan of the underlying documents which is likely to be inefficient in our scenario. Our experimental results in Sect. 7 show that our PDT generation is more than an order of magnitude faster then PROJ.

Our idea also appears similar to creating query-equivalent documents (QED for short) proposed in [8]. Querying over the PDTs produces the same sequence of result elements as querying over the original documents except that our results will be materialized separately; hence the PDTs are query-equivalent to the underlying documents with respect to the view query. There are, however, two key differences between the construction algorithms. First, QED performs the DFS of the structural pattern (similar to QPT) and exploits structural joins to create the corresponding document structure. This is similar to [13,33], and likely to be costly in our scenario. Second, QED retrieves element contents (in the Pruning algorithm) or the entire subtree (in the Lopping algorithm) during query processing and fully materializes all result elements. This is similar to [28] and is unnecessarily expensive since only the top-K results need to be fully materialized.

We now illustrate more details of PDTs before presenting our algorithms.

4.1 PDT illustration and definition

The key idea of a PDT is that an element *e* in the document corresponding to a node *n* in the QPT is selected for inclusion only if it satisfies all three types of constraints: (1) an ancestor constraint, which requires that an ancestor element of *e* that, if *e* is not the root node, corresponds to the parent of *n* in the QPT should also be selected, (2) a descendant constraint, which requires that for each mandatory edge from *n* to a child of *n* in the QPT, at least one child/descendant element of *e* corresponding to that child of *n* should also be selected, and (3) a predicate constraint, which requires that if *e* is a leaf node, it satisfies all predicates associated with *n*. Consequently, there is a mutual restriction between ancestor and descendant elements. In our example, only reviews with at least one isbn subelement are selected (due to the descendant constraint), and only those isbn and content elements that have a selected review are selected (due to the ancestor constraint). Note that this restriction is not "local": a content element is not selected for a review if that review does not contain an isbn element.

We now formally define notions of PDTs. We first define the notion of *candidate elements* that only captures descendant restrictions.

Definition 1 (*candidate elements*) Given a QPT Q, an XML document D, the set of candidate elements in D associated with a node $n \in Q$, denoted by $CE(n, D)$, is defined recursively as follows.

– *n is a leaf node in Q:* CE(n, D) =
  {v ∈ D | tag name of v is n.tag ∧
  the value of v satisfies all predicates in n.preds }.
– *n is a non-leaf node in Q:* CE(n, D) =
  {v ∈ D | tag name of v is n.tag ∧ for every edge e in Q,
  if e.parent is n and e.ann is 'm' (mandatory),
  then ∃ec ∈ CE(e.child, D) such that
  (a) e.axis = '/' and v is the parent of ec, or
  (b) e.axis = '//' and v is an ancestor of ec }

Definition 1 recursively captures the descendant constraints from bottom up. For example, in Fig. 6a, candidate elements corresponding to "review" must have a child element "isbn". Now we define notions of *PDT elements* which capture both ancestor and descendant constraints.

Definition 2 (*PDT elements*) Given a QPT Q, an XML document D, the set of PDT elements associated with a node $n \in Q$, denoted by *PE(n, D)*, is defined recursively as follows.

– *n is the root node of Q:* PE(n, D) = CE(n, D)
– *n is the non-root node in Q:* PE(n, D) =
  {v ∈ D | v is in CE(n, D) ∧
  for every edge e in Q, if e.child is n,
  then $\exists vp$ ∈PE(e.parent, D) such that
  (a) e.axis = '/' and vp is the parent of v, or
  (b) e.axis = '//' and vp is an ancestor of v}

Intuitively, the PDT elements associated with each QPT node are first the corresponding candidate elements and hence satisfy descendant constraints. Further, the PDT elements associated with the root QPT node are just its candidate elements, because the root node does not have any ancestor constraints; the PDT elements associated with a non-root QPT node have the additional restriction that they must have the parent/ancestors that are PDT elements associated the parent QPT node. For example, in Fig. 6a, each PDT element corresponding to "content" must have a parent element that is the PDT element with respect to "review". Using the definition of PDT elements, we can now formally define a PDT.

Definition 3 (*PDT*) Given a QPT Q, an XML document D, and a set of keywords K, a PDT is a tree (N, E) where N is the set of nodes and E is set of edges, which are defined as follows.

– N = $\cup_{q \in Q}$ PE(q, D), and nodes in N are associated with required values, tf values and byte lengths.
– E = {(p, c) | p, c are in N ∧ p is an ancestor of c ∧ $\nexists q \in N$ s.t. p is an ancestor of q and q is an ancestor of c}

4.2 Proposed algorithms

We now propose our algorithm for efficiently generating PDTs. The generated PDTs satisfy all restrictions described above and contains selectively materialized element values. The main feature of our algorithm is that it issues index lookups whose number is in proportion to the size of the query, not the size of the underlying data, and only makes a single pass over the relevant path and inverted lists indices.

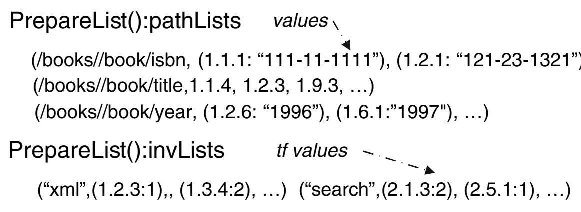
At a high level, the development of the algorithm requires solving three technical problems. First, how do we minimize the number of index accesses? Second, how do we efficiently materialize required element values? Finally, how do we efficiently generate the PDTs using the information gathered from indices? We describe our solutions to these problems in turn in the next two sections.

4.2.1 Optimizing index probes and retrieving join values

To retrieve Dewey IDs and element values required in PDTs, our algorithm invokes a query-dependent number of probes on path indices. First, we issue index lookups for QPT nodes that do not have mandatory child/descendant edges; note that this includes all the leaf nodes. The elements corresponding to these nodes could be part of the PDT even if none of its descendants are present in the PDT according to the definition of mandatory edges [13]. Further, if a QPT node is associated with predicates, the index lookup will only return elements that satisfy the predicates. For instance, for the book QPT shown in Fig. 6a, we only need to perform three index

```
1:  PrepareLists (QPT $qpt$, PathIndex $pindex$, InvertedIndex
    $iindex$, KeywordSet $kwds$): (PathLists, InvLists)
2:    $pathLists \leftarrow \emptyset$; $invLists \leftarrow \emptyset$
3:    for Node $n$ in $qpt$ do
4:      $p \leftarrow PathFromRoot(n)$; $newList \leftarrow \emptyset$
5:      if n has a 'v' annotation then
6:        {Combining retrieval of IDs and values}
7:        $newList \leftarrow (n, pindex.LookUpIDValue(p))$
8:        if n has no mandatory child/descendant edges then
9:          $newList \leftarrow (n, pindex.LookUpID(p))$
10:       end if
11:     end if
12:     if newList $\neq$ null then  $pathLists.add(newList)$
13:   end for
14:   for all k in kwds do
15:     $invLists \leftarrow invLists \cup (k, sindex.lookup(k))$
16:   end for
17:   return (pathLists, invLists)
```

Fig. 7 Retrieving IDs and values


PrepareList():pathLists values

(/books//book/isbn, (1.1.1: "111-11-1111"), (1.2.1: "121-23-1321"),...)
(/books//book/title,1.1.4, 1.2.3, 1.9.3, …)
(/books//book/year, (1.2.6: "1996"), (1.6.1:"1997"), …)

PrepareList():invLists tf values

("xml",(1.2.3:1),, (1.3.4:2), …) ("search",(2.1.3:2), (2.5.1:1), …)

Fig. 8 Results of PrepareLists()

lookups on path indices (shown in Fig. 5) for three paths in QPT: */books//book/isbn*, */books//book/year[.>1995]*, and */books//book/title*.

Second, for nodes with 'v' annotation, we issue separate lookups to retrieve their data values (which may be combined with the first round of lookups). The idea of retrieving values from path indices is inspired by a simple, yet important observation that path indices already store element values in (Path, Value) pairs. Our algorithm conveniently propagates these values along with Dewey IDs. For example, consider the QPT of the book document in Fig. 6a and the path indices in Fig. 5. For the path */books//book/isbn*, we use its path to look up the B+-tree index over (Path, Value) pairs in the *Path-Values* table to identify all corresponding values and Dewey IDs (this can be done efficiently because Path is the prefix of the composite key, (Path, Value)); in Fig. 5, we would retrieve the second and third rows from the *Path-Values* table. Note that IDs in individual rows are already sorted. We then merge the ID lists in both rows and generate a single list ordered by Dewey IDs, and also associate element values with the corresponding IDs. For example, the Dewey ID 1.1.1 will be associated with the value "111-111-1111".

Finally, for each query keyword, our algorithm returns the relevant inverted index indices to obtain scoring information. For example, for the keyword *xml*, the returned inverted list contains Dewey ID 1.2.3 with its term frequency of *xml*.

Figure 7 shows the high-level pseudo-code of our algorithm of retrieving Dewey IDs, element values and tf values. The algorithm takes a QPT, Path Index, query keywords,

```
1:  GeneratePDT (QPT $qpt$, PathIndex $pindex$, KeywordSet
    $kwds$, InvertedIndex $iindex$): PDT
2:    $pdt \leftarrow \emptyset$
3:    (pathLists, invLists) $\leftarrow$ PrepareLists(qpt, pindex,
      iindex, kwds)
4:    for idlist $\in$ pathLists do
5:      AddCTNode(CT.root, GetMinEntry(idlist), 0)
6:    end for
7:    while CT.hasMoreNodes() do
8:      for all n $\in$ CT.MinIDPath do
9:        q $\leftarrow$ n.QPTNode
10:       if pathLists(q).hasNextID() $\wedge$ there do not exist
          $\geq$ 2 IDs in pathLists(q) and also in CT then
11:         AddCTNode(CT.root, pathLists(q).NextMin(),
            0)
12:       end if
13:     end for
14:     CreatePDTNodes(CT.root, qpt, pdt)
15:   end while
16:   return pdt
```

Fig. 9 Algorithm for generating PDTs

and Inverted Index as input, and first issues a path indices lookup on nodes that have a 'v' annotation (lines 5–10), to obtain the values and IDs. It then looks up path indices for each QPT node that has no mandatory child/descendant edges to obtain just Dewey IDs (lines 8–10). Finally, the algorithm looks up inverted lists indices and retrieves the list of Dewey IDs containing the keywords along with tf values (lines 14–16). Figure 8 shows the output of PrepareList for the book QPT (Fig. 6a). Note that the ID lists corresponding to */books//book/isbn* and */books//book/year* contain element values, and the ID lists retrieved from inverted lists indices contain tf values.

4.2.2 Efficiently generating PDTs

In this section, we propose a novel algorithm that makes a single "merge" pass over the lists produced by PrepareList and produces the PDT. The PDT satisfies the ancestor/descendant constraints (determined using Dewey IDs in pathLists) and contains selectively materialized element values (obtained from pathLists) and tf values w.r.t each query keyword (obtained from invLists). For our running example, our algorithm would produce the PDT shown in Fig. 6b by merging the lists shown in Fig. 8.

The main challenges in designing such an algorithm are (1) we must enforce complex ancestor and descendant constraints (described in Sect. 4.1) by scanning the lists of Dewey IDs only once, (2) ancestor/descendant axes may expand to full paths consisting of multiple IDs matching the same QPT nodes, which adds additional complication to the problem.

The key idea of the algorithm is to process ids in Dewey order. By doing so, it can efficiently check descendant restrictions because all descendants of an element will be clustered immediately after that element in pathLists. Figure 9 shows the high-level pseudo-code of our algorithm which works as follows. The algorithm takes in a QPT, path index, and inverted index of the document, and begins by invoking

PrepareList in order to collect the ordered lists of ids relevant to the view. It then initializes the *Candidate Tree* (described in more detail shortly) using the minimum ID in each list (lines 4–6). Next, the algorithm makes a single loop over the IDs in pathLists (lines 7–15), and creates PDT nodes using information stored in the CT. At each loop, the algorithm processes and removes the element corresponding to the minimum ID in the CT. Before processing and removing the element, it adds the next ID from the corresponding path list (lines 8–12) so that we maintain the invariant that there is at least one ID corresponding to each relevant QPT node for checking descendant constraints.

Next the algorithm invokes the function CreatePDTNodes (line 14) and checks if the minimum element satisfies both ancestor and descendant constraints. If it does, we will create it in the result PDT. If it satisfies only descendant constraints, we store it in a temporary cache (PdtCache) so that we can check the ancestor constraints in subsequent loops. If it does not satisfy descendant constraints and does not have any children in the current CT, we discard it immediately. The intuition is that in this case, since the CT already contains at least one ID for each relevant QPT node (by the invariant above), and since IDs are retrieved from pathList in Dewey order, we can infer that the minimum element cannot have any unprocessed descendants in pathLists; hence it will not satisfy descendant constraints in all subsequent loops. The algorithm exits the loop and terminates after exhausting IDs in pathList and the result PDT contains all and only IDs that satisfy the PDT definition.

Description of the candidate tree: The Candidate Tree, or the CT, is a tree data structure. Each node *cn* in the CT stores sufficient information for efficiently checking ancestor and descendant constraints and has the following five components:

- ID: the *unique* identifier of *cn*, which always corresponds to a prefix of a Dewey ID in pathLists.
- QNode: the QPT node to which cn.ID corresponds.
- ParentList (or PL): a list of cn's ancestors whose QNode's are the parent node of cn.QNode.
- DescendantMap (or DM):*QNode*→ *bit*: a mapping containing one entry for each mandatory child/descendant of cn.QNode. For a child QPT node c, DM[c] = 1 *iff* cn has a child/descendant node that is a candidate element with respect to c.
- PdtCache: the cache storing cn's descendants that satisfy descendant restrictions but whose ancestor restrictions are yet to be checked.

We now illustrate these components using CT shown in Fig. 12a, which is created using IDs 1.1.1, 1.1.4, and 1.2.6, corresponding to paths in pathLists shown in Fig. 8. First,

```
1:  AddCTNode(CTNode parent, DeweyID id, int depth)
2:    newNode ← null
3:    if depth ≤ id.Length then
4:      curId←Prefix(id, depth); qNode←QPTNode(curId)
5:      if qNode = null then
        AddCTNode(parent,id,depth+1)
6:      else
7:        newNode ← parent.findChild(curId)
8:        if newNode = null then
9:          newNode ← parent.addChild(curId, qNode)
10:         Update the data value and tf values if required
11:       end if
12:       AddCTNode(newNode, id, depth+1)
13:     end if
14:   end if
15:   if newNode≠null ∧ ∀i, newNode.DM[i]=1 then
16:     ∀ n∈newNode.PL, n.DM[newNode.QPTNode]←1
17:   end if
```

Fig. 10 Algorithm for adding new CT nodes

every node has an ID and a QNode and CT nodes are ordered based on their IDs. For example, the ID of the "books" node is 1 which corresponds to a prefix of the ID 1.1.1, and the id 1.1.1 corresponds to the QPT node "isbn". The PL of a CT node stores its ancestor nodes that correspond to the parent QPT node. For instance, book1.PL = {books}. Note that cn.PL may contain multiple nodes if cn.QNode is in an ancestor/descendant relations. For example, if "/books//-book" expands to "/books/books/book", then book.PL would include both "books". Next, DM keeps track of whether a node satisfies descendant restrictions. For instance, book1.DM[year] = 0 because it does not have the mandatory child element "year" while book2.DM[year] = 1 because it does. Consequently, a CT node satisfies the descendant restrictions (and therefore is a *candidate element*) when its DM is empty (corresponding to QPT nodes without mandatory child edges), or the values in its DM are all 1 (corresponding to QPT nodes with mandatory child edges). PdtCache will be illustrated in subsequent steps shortly. Note that for ease of exposition, our illustration focuses on creating the PDT hierarchy; the atomic values and tf values are not shown in the figure but bear in mind that they will be propagated along with Dewey IDs.

We now describe the Candidate Tree and individual steps of the algorithm in more detail.

Initializing the candidate tree: As mentioned earlier, the algorithm begins by initializing the CT using minimum IDs in pathLists. Figure 10 shows the pseudo-code for adding a single Dewey ID and its prefixes to the CT. A prefix is added to the CT if it has a corresponding QPT node and is not already in the CT (lines 6–13). In addition, if a prefix is associated with a 'c' annotation, the tf values are retrieved from the inverted lists (line 10).

Figure 12a, which we just described, shows the initial CT for our example, which is created by adding minimum IDs of paths in pathLists shown in Fig. 8. Note that for ease of exposition, our algorithm assumes each Dewey ID corresponds to

```
1:  CreatePDTNodes (CTNode n, QPT qpt, PDT
    parentPdtCache)
2:    if ∀i, n.DM[i] = 1 ∧n.ID not in parentPdtCache then
3:      pdtNode = parentPdtCache.add(n)
4:    end if
5:    if n.HasChild() = true then
6:      CreatePDTNodes(n.MinIdChild, qpt, n.PdtCache)
7:    else
8:      {Handle pdt cache and then remove the node itself}
9:      for x in n.pdtCache do
10:       {Update parent list and then propagate x to
          parentPdtCache}
11:       if n ∈ x.PL then
12:         x.PL.remove(n)
13:         if ∃i, n.DM[i] = 0 ∧ x.PL = ∅ then
            n.pdtCache.remove(x)
14:         else
15:           x.PL.replace(n, n.PL)
16:         end if
17:       end if
18:       if x ∈ pdtCache then  Propagate x to
          parentPdtCache
19:     end for
20:     n.RemoveFromCT()
21:   end if
```

Fig. 11 Processing CT.MinIDPath

a single QPT node; however, when the QPT contains repeating tag names, one Dewey ID can correspond to multiple QPT nodes. We discuss how to handle this case in Sect. 5.

Description of the main loop: Next the algorithm enters the loop(lines 7–15 in Fig. 9) which adds new Dewey IDs to the CT and creates PDT nodes using CT nodes. At each loop, the algorithm ensures the following invariant: the Dewey IDs that are processed and known to be PDT nodes are either in the CT or in the result PDT (hence we do not miss any potential PDT nodes); and the result PDT only contains IDs that satisfy the PDT definition.

As mentioned earlier, at each loop we focus on the element corresponding to the minimum ID in the CT and its ancestors (denoted by MinIDPath in the algorithm). Specifically, we first retrieve next minimum IDs corresponding to QPT nodes in MinIDPath(Step 1). We then copy IDs in MinIDPath from top down to the result PDT or the PDT cache (Step 2). Finally, we remove those nodes in MinIDPath that do not have any children (Step 3). We now describe each step in more detail.

Step 1 (*adding new IDs*) In this step, the algorithm adds the current minimum IDs in pathLists corresponding to the QPT nodes in CT.MinIDPath. In Fig. 12a, this path is "/books //book/isbn" and Fig. 12b shows the CT after its next minimum ID 1.2.1 is added (for reason of space, this figure and the rest only show the QPT node and ID).

Step 2 (*creating PDT nodes*) In this step, the algorithm creates PDT nodes using CT nodes in CT.MinIDPath from top down (Fig. 11, lines 2–4). We first check if the node satisfies the descendant constraints using values in its DM. In Fig. 12b, DM of the element "books" has value 1 in all entries; hence

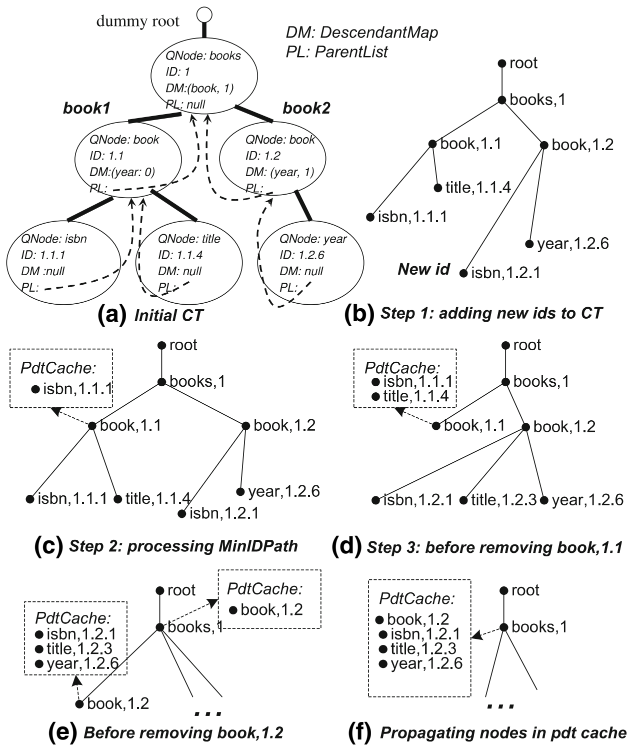

dummy root
DM: DescendantMap
PL: ParentList
QNode: books ID: 1 DM:(book, 1) PL: null
book1
book2
QNode: book ID: 1.1 DM:(year: 0) PL:
QNode: book ID: 1.2 DM: (year, 1) PL:
QNode: isbn ID: 1.1.1 DM :null PL:
QNode: title ID: 1.1.4 DM: null PL:
QNode: year ID: 1.2.6 DM: null PL:
(a) Initial CT
root
books,1
book,1.1
book,1.2
title,1.1.4
isbn,1.1.1
year,1.2.6
New id
isbn,1.2.1
(b) Step 1: adding new ids to CT
PdtCache: isbn,1.1.1
root
books,1
book,1.1
book,1.2
isbn,1.1.1
title,1.1.4
year,1.2.6
isbn,1.2.1
(c) Step 2: processing MinIDPath
PdtCache: isbn,1.1.1 title,1.1.4
root
books,1
book,1.1
book,1.2
isbn,1.2.1
title,1.2.3
year,1.2.6
(d) Step 3: before removing book,1.1
root
PdtCache: book,1.2
PdtCache: isbn,1.2.1 title,1.2.3 year,1.2.6
books,1
book,1.2
(e) Before removing book,1.2
PdtCache: book,1.2 isbn,1.2.1 title,1.2.3 year,1.2.6
root
books,1
(f) Propagating nodes in pdt cache

Fig. 12 Generating PDTs

we will create its ID in the PDT cache passed to it(lines 2–4), which is the result PDT.

The algorithm then recursively invokes CreatePDTNodes on the element *book1* (line 6). Its DM has value 0 and hence it is not a PDT node *yet*. Next, we find its child element "isbn" has an empty DM and satisfies the descendant restrictions. Hence we create the node "isbn" in book1.PdtCache. Figure 12c illustrates this step. In general, the pdt cache of a CT node stores the ids of descendants that satisfy the descendant restrictions; ancestor restrictions are only checked when the CT node is removed (in Step 3).

Step 3 (*removing CT nodes*) After the top down processing, the algorithm starts removing nodes from bottom up (Fig. 11, line 7–29). For instance, in Fig. 12c, after we process and remove the node "title", we will remove the node "book" because it does not have children and it does not satisfy descendant constraints. Figure 12d shows the CT at this point. Note that since we process nodes in id order, we can infer that the descendant constraints of this node will never be satisfied in the future.

Another key issue we consider before removing a node is to handle nodes in its pdt cache. In our example, the pdt cache contains two nodes "isbn" and "title". As mentioned earlier, they both satisfy descendant constraints. Hence we only need to check if they satisfy ancestor constraints, which is done by checking nodes in their parent lists. If those parent nodes are known to be non-PDT nodes, which is the case for “isbn” and “title”, then we can conclude the nodes in the cache will not satisfy ancestor restrictions, and can hence be removed (line 13). Otherwise the cache node still has other parents, which could be PDT nodes, and will be propagated to the pdt cache of the ancestor. Fig. 6e and f illustrates this case in our running example, which occurs when we remove the node “book” with ID 1.2.

Finally, at the last step of the algorithm when we remove the root node “books”, all IDs in its pdt cache will be propagated to the result PDT. In summary, we remove a node (and its ID) only when it is known to be a non-PDT node, which is either a CT node that does not satisfy descendant constraints, or a node in a pdt cache that does not satisfy ancestor constraints. Further, we only create nodes satisfying descendant constraints in the pdt cache, and always check ancestor constraints before propagating them to ancestors in the CT. Therefore it is easy to verify the invariant of the main loop holds.

Scoring & generating the results. As mentioned in Sect. 3.1, our system transforms the original fn:doc() which reads input document to fn:pdt() which reads the PDT, and transforms the ftcontain() in the traditional evaluator to our customized Scoring & Materialization Module. The remaining part of the evaluator is reused to handle the query processing to create the projected view. This is shown in Fig. 3. Once the PDTs are generated (e.g., the PDT of our running example is shown in Fig. 6b), they are fed to a traditional evaluator to produce the temporary results. which are then sent to the Scoring & Materialization Module. Using just the pruned results with required tf values and byte lengths (encoded as XML attributes as shown in Fig. 6b), this module first enforces conjunctive or disjunctive keyword semantics by checking the tf values, and then computes scores of the view results. Specifically, for a view result s, score(s) is computed as follows: first calculate $tf(s, k)$ for a keyword k by aggregating values of $tf(s', k)$ of all relevant base elements s'; then calculate the value $idf(k)$ by counting the number of view results containing the keyword k; next use the formula in Sect. 2.2 to obtain the non-normalized scores, which are then normalized using aggregate byte lengths of the relevant base elements.

The Scoring & Materialization Module then identifies the ordered view results with top-K scores. Only after the final top-K results are identified are the contents of these results retrieved from the document storage system; consequently, only the contents required for producing the results are retrieved.

5 Generalizations and optimizations

In this section, we describe substantial generalizations and optimizations to the algorithms proposed in Sect. 4. The generalized algorithms handle a larger class of complex XML views that include descendant axes, element wild cards, and repeating element tag names.

If an XML view definition contains descendant axes and element wild cards, our path indices will expand them to possibly multiple full paths (Sect. 3.2). The main challenge in this case is that, a single Dewey ID can match multiple QPT nodes. Further, different QPT nodes capture different ancestor/descendant constraints. Hence they must be treated separately. For example, if the QPT path is “//a //a” and the corresponding full data path is “/a /a /a”, then the second “a” in the full path matches both nodes in the QPT path. To handle this case, we extend the structure of CT node to contain a set of QNodes, each of which is associated with their own InPdt, PL and DM.

Further, the current algorithm always copies IDs that satisfy the descendant constraints in the pdt cache. This can be optimized by immediately creating the IDs in the result PDT if they also satisfy the ancestor restrictions. For this purpose, we add a boolean flag InPdt to the CT node, set InPdt to be true when the ID is created in the result PDT, and create the descendant ID in the PDT when one of its parents is in the PDT (InPdt = true).

Now we describe our algorithms that handle these extensions and optimizations in more detail.

Description of the algorithm: Figure 13 shows the high-level pseudo-code of our generalized and optimized algorithm. The algorithm takes in a QPT, path index, and inverted index of the document, and generates the PDT. It begins by invoking PrepareList() to collect the lists of ids relevant to the view, and then initializes the *Candidate Tree* using the minimum Dewey ID in each list (lines 6–8).

Now we focus on the generalization part of the algorithm.

Extension of candidate tree: As mentioned earlier, when QPT contains repeating tag names and/or descendant axes, one document node can correspond to multiple QPT nodes. In these cases, we have to store all of the corresponding QPT nodes because in general, different QPT nodes capture different ancestor and descendant restrictions. To handle this in CT, we map each CT node to a set of QPT nodes, namely CTQNodeSet. Since the notion of parent and descendants are associated with each QPT node, we also remove them from the top-level structure of CT node. Formally, each node *cn* in CT has the following three components:

- ID: the *unique* identifier of *cn*, which always corresponds to a prefix of a Dewey ID in pathLists.
- CTQNodeSet: the set of QPT nodes to which cn.ID corresponds.
- PdtCache: the cache storing cn’s descendants that satisfy descendant restrictions but whose ancestor restrictions are yet to be checked.

```
1:  GeneratePDT (QPT $qpt$, PathIndex $pindex$, KeywordSet
    $kwds$, InvertedIndex $iindex$): PDT
2:    $pdt \leftarrow \emptyset$
3:    (pathLists, invLists) $\leftarrow$ PrepareLists(qpt, pindex,
      iindex, kwds)
4:    {Initialize CT}
5:    for idlist $\in$ pathLists do
6:      AddCTNode(CT.root, GetMinEntry(idlist), 0)
7:    end for
8:    while CT.hasMoreNodes() do
9:      {Adding ids corresponding to the left most path}
10:     lmp $\leftarrow$ CT.LeftMostPath
11:     for all cqn $\in$ lmp do
12:       for all qn in cqn.CTQNodes where $\exists l \in$ pathLists,
          l.QNode = cqn do
13:         if curList.hasNextID() then
14:           AddCTNode(CT.root,
              curList.GetNextMinEntry(), 0)
15:         end if
16:       end for
17:     end for
18:     CreatePDTNodes(CT.root, qpt, pdt, pdt)
19:   end while
20:   return pdt
```

Fig. 13 Algorithm for generating PDTs

```
1:  AddCTNode(CTNode $parent$, DeweyID $id$, int depth)
2:    if depth $\leq$ id.Depth then
3:      curId $\leftarrow$ Prefix(id, depth); qNodes $\leftarrow$ QNodes(curId)
4:      if qNodes = $\emptyset$ then AddCTNode(parent,id,depth+1)
5:      else
6:        newNode $\leftarrow$ parent.findChild(curId)
7:        if newNode = null then
8:          newNode $\leftarrow$ parent.addChild(curId, qNodes)
9:          Initialize newNode.CTQNodeSet using qNodes
10:         Update the data value and tf values if required
11:       end if
12:     end if
13:     AddCTNode(newNode, id, depth+1)
14:   end if
15:   for all q in qNodes do
16:     if $\forall i$, q.DM[i]=1 then
17:       set DM[q] to 1 for nodes in q.PL
18:     end if
19:   end for
```

Fig. 14 Algorithm for adding new CT nodes

Each node in CTQNodeSet has the following members:

– QNode: one of the QPT nodes to which cn.ID corresponds.
– ParentList (or PL): a list of cn's ancestors whose QNode's are the parent node of cn.QNode.
– DescendantMap (or DM):*QNode*→ *bit*: a mapping containing one entry for each mandatory child/descendant of cn.QNode. For a child QPT node c, DM[c] = 1 *iff* cn has a child/descendant node that is a candidate element with respect to c.
– InPdt: a boolean flag indicating whether cn ∈ PE(q, D);

Figure 16b illustrates the new structure of a CT node that will be used later when we walk through the algorithm.

Generalization and optimization of the algorithm: For completeness we include the full algorithms in Figs. 13, 14

```
1:  CreatePDTNodes (CTNode n, QPT $qpt$, PDT $pdt$, PDT
    $parentPdtCache$)
      {Create PDT nodes using CT nodes in left most path}
2:    for all q in n.CTQNodes where q.InPdt = false do
3:      if $\forall i$, q.DM[i] = 1 then
4:        if q.PL = $\emptyset$ $\vee$ $\exists$ p $\in$ q.PL, p.InPdt = true then
5:          q.InPdt = true; Write n.Id to pdt if n.id $\notin$ pdt
6:        else
7:          pdtCacheNode = parentPdtCache.find(n.Id)
8:          if parentCacheNode = null then pdtCacheNode
            = parentPdtCache.add(n.Id)
9:          for all q in n.CTQNodes where $\forall i$, q.DM[i] = 1
            do
10:           pdtCacheNode.PL.add(q.PL)
11:         end for
12:       end if
13:     end if
14:   end for
15:   if n.HasChild() = true then
16:     {Recursively handle the left most child(LMC)}
17:     CreatePDTNodes(LMC, qpt, pdt, n.PdtCache)
18:   else
19:     {Handle pdtCache and then remove the node itself}
20:     for x in n.pdtCache do
21:       if x.PLx = $\emptyset$ $\vee$ $\exists p \in$ x.PL, p.InPdt = true then
          Write x.id to pdt if $x.id \notin$ pdt
22:       else
23:         {Update parent list and then propagate x to
            parentPdtCache}
24:         for all q in n.CTQNodes where q in PL(x) do
25:           x.PL.remove(q)
26:           if $\exists i$, q.DM[i] = 0 $\wedge$ PL(x) = $\emptyset$ then
              n.pdtCache.remove(x)
27:           else
28:             x.PL.replace(q, q.PL)
29:           end if
30:         end for
31:         if x $\in$ pdtCache then PropagatePDT(x,
            parentPdtCache)
32:       end if
33:     end for
34:     n.RemoveFromCT()
35:   end if
```

Fig. 15 Algorithm for generating PDTs

and 15. The main generalization lies in checking all nodes in CTQNodeSet for structural constraints. The main optimization is achieved by directly creating qualified CT nodes in PDT and hence bypassing the unnecessary use of pdt cache. Specifically, in Fig. 15, lines 2–5, the algorithm processes the CT nodes in the left most path from top down. If a CT node n in the left most path has an item q ∈ n.CTQNodeSet s.t. q.DM has the value 1 in all entries, then we know that q satisfies the descendant restrictions. Further, if q.PL = ∅ or $\exists p$ ∈ q.PL, q.InPdt = true, then we can infer that q also satisfies the ancestor restrictions. Therefore we immediately create the corresponding node in the result PDT.

Now we walk through the algorithm using the QPT and ID lists shown in Fig. 16a to illustrate the main generalizations and optimizations. The algorithm first initializes CT by merging the minimum Dewey IDs 1.1.1.1, 1.1.1.2 and 1.3.1.3 corresponding to the full path "a /x /b /c", "a /x /b /d" and "a /x /b /e" respectively (lines 6–8). In **AddNewCTNode**(), CT nodes are created for distinct Dewey ID components that match a QPT node. Note the IDs corresponding to the tag "x" is pruned from the CT because they are not relevant to our view, and the structural relations of other nodes

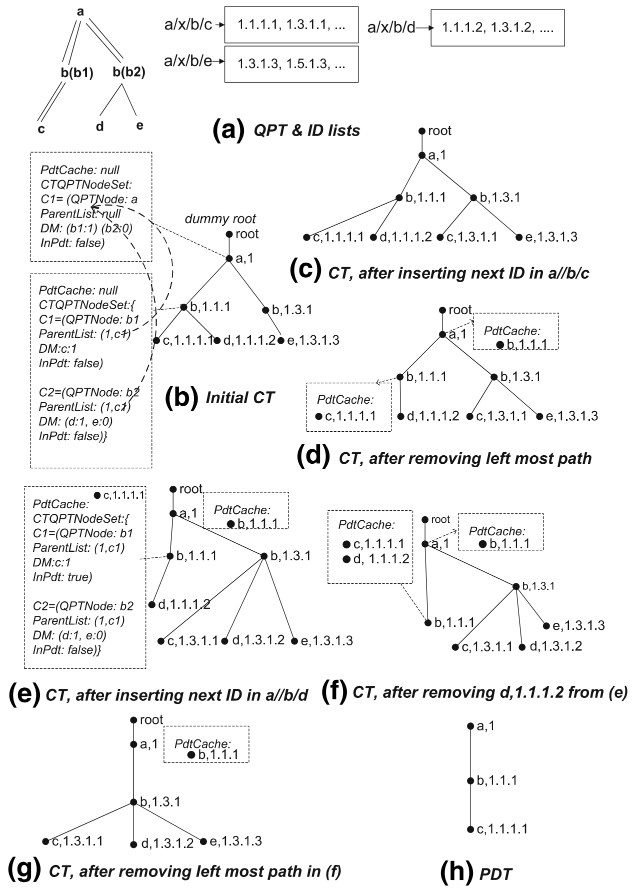


Fig. 16 Generating PDT

can still be easily determined using their respective Dewey IDs.

Further, as mentioned earlier, new nodes will update the Descendant Map DM of nodes in their PL. Figure 16b shows the initial CT. Note for reason of space, we only show the full content of the CT node for nodes a(1) and b(1.1.1). As shown, the element $b(1.1.1)$, which corresponds to the QPT nodes $b1$ and $b2$ (in Fig. 16a), contains two items $c1$ and $c2$ in its CTQNodeSet. c1.DM[c] = 1 because there is a child nodes $c(1.1.1.1)$ corresponding to the QPT node c; however, c2.DM[e] = 0 because it does not have a child node corresponding to the QPT node e. Also, due to this, in the root element $a(1)$, c1.DM[b2] = 0. Note that at this point the invariant holds because all IDs are in the CT and the result PDT is empty.

After the CT is initialized, the algorithm begins creating the result *PDT* by repeatedly invoking **CreatePDTNodes** (lines 9–20). As mentioned earlier, it first retrieves next IDs corresponding to the left most path "a//b/c". Figure 16c shows the content of the CT at this point. The algorithm then inspects CT nodes from top down on the current leftmost path because it most likely contains the nodes that are known to be PDT or non-PDT nodes with minimum IDs. In our example, it first inspects the root node $a(1)$ (lines 2–4) and determines it is not a PDT node (yet) because all items in its CTQNodesSet do not satisfy the descendant restrictions (DM's have the value 0). Hence we just recursively call **CreatePDTNodes** on the leftmost child $b(1.1.1)$ (line 17). The item c1 in this node satisfies the descendant restrictions (DM has the value 1 in all entries), but none of its parent is in the pdt. Hence it is not known whether it should indeed be included in the result PDT. Thus we temporarily create it in the pdt cache of its parent node, a(1). We similarly handle the node (c, 1.1.1.1), and then remove it from the CT because it is a leaf node. Figure 16d shows the content of the CT after this step. Since we keep all the IDs in the candidate tree, the invariant still holds.

Now we enter the next loop and add the next minimum ID corresponding to a//b/d, which is 1.3.1.2. Figure 16e shows the content of the CT. Note after adding this id, the node a(1) now satisfies the descendant restriction because the node b(1.3.1) is the candidate element of both QPT nodes b1 and b2. Hence in the second phase, we will write the id 1 to the result PDT. And similarly, we will write id 1.1.1 to the result PDT. Next we arrive at the node d(1.1.1.2). Since its parent item is c2 in the node b(1.1.1) and c2.inPdt = false, which implies that the structural restrictions corresponding to c2.QNode are not satisfied. Therefore we cannot write the id 1.1.1.2 to the result PDT even though the parent id 1.1.1 is in the PDT. Hence we write this ID to the pdt cache of the node b(1.1.1). Figure 16g shows the content of the CT. We then remove the node 1.1.1.2 since it is a leaf node. Next we will remove the node b(1.1.1). First we check nodes in its pdt cache. There are two items, c(1.1.1.1) and d(1.1.1.2). We will write c(1.1.1.1) to the result PDT because its parent is the item c1 and c1.inPdt = true; however, we will not output the node d(1.1.1.2) because its parent c2 does not and will not satisfy the descendant restrictions. This illustrates how the mutual restrictions are enforced. Figure 16f shows the content of the PDT at this point. It is easy to verify that at this point the invariant still holds. The IDs that satisfy the PDT definition are 1, 1.1.1, 1.1.1.1, 1.3.1, 1.3.1.1, 1.3.1.2, and 1.3.1.3. The first three are in the result PDT, and the rest are in the CT. And the result PDT only contains the first three IDs.

5.1 Complexity of algorithms

The runtime of GeneratePDT is $O(Nqdf + Nqd^2 + Nd^3 + Ndkc)$ where N is the number of IDs in pathLists, d is the depth of the document, q and f are the depth and maximal fan-out of the QPT, respectively, k is the number of keywords, and c is the average unit cost of retrieving tf values. Intuitively, the top-down and bottom-up processing dominate the overall cost. $Nqdf + Nqd^2$ determines the cost of the top-down processing: there can be Nd ID prefixes; every prefix can correspond to q QPT nodes; every QPT node can have d parent CT nodes and f mandatory child nodes. Nd^3 determines the cost of bottom-up processing, since every prefix can be propagated d times and can have d nodes in its parent list. Finally, $Ndkc$ determines the cost for retrieving tf values from the inverted index.

Note that this is a worst case bound which assumes multiple repeating tags in queries (q QPT nodes), and repeating tags in documents (d parent nodes). In most real-life data, these values are much smaller (e.g., DBLP,[4] SIGMOD Record,[5] and INEX), as also seen in our experiments.

6 Correctness of GeneratePDT

We can prove the following correctness theorem:

Theorem 1 (Correctness) *Given a set of keywords KW, an XQuery query Q and a database D ∈ UD, if PDTDB = {GeneratePDT(QPT, D.PathIndex, D.InvertedIndex, KW) | QPT ∈ GenerateQPT(Q) } , then*

– *I(Q(PDTDB)) = Q(D)(The result sequences, after being transformed, are identical)*
– *∀e ∈ Q(PDTDB), e′ ∈ Q(D), I(e) = e′ ⇒ PDTByteLength(e) = len(e′) (The byte lengths of each element are identical)*
– *∀e ∈ Q(PDTDB), e′ ∈ Q(D), I(e) = e′ ⇒ (∀k ∈ KW, PDTTF(e,k) = tf(e′,k)) (The term frequencies of each keyword in each element is identical)*

Note that in Theorem 1, I is the function transforming Dewey IDs to node contents, PDTTF is the tf calculation function, and PDTByteLength is the byte length calculation function, len(e) is the byte length of a materialized element e, and using the notations of UD, Q, S defined in Sect. 2.1.

To prove Theorem 1, it suffices to show the correctness of the equivalence of the query results; when this holds, the equivalence of byte lengths and term frequencies are obvious since the algorithm merely propagates these values and does simple aggregation during element constructions.

The full proof of the equivalence of the query results is quite complex and we refer the reader to Appendix A for details. Here we just briefly describe some intuitions.

First, as shown in [13] and Sect. 4.1 PDTs by definition are query-equivalent to the original documents. Hence it suffices to show that GeneratePDT produces PDTs that satisfy the definitions. In other words, we need to show that the result PDTs contain and only contain Dewey IDs that satisfy all structural constraints.

[4] http://dblp.uni-trier.de/xml/.

[5] http://acm.org/sigmod/record/xml/.

```
let $view :=
for $aurthor in fn:doc(authors.xml)/author
return <author_articles>
        <author> {$author//fnm},{$author//snm} </author>,
        {for $article in fn:doc(*)/books/journal/article
         where $article//au//fnm = $author/fnm
          and $article//au//snm = $author/snm
         return $article}
    </author_articles>
```

Fig. 17 INEX view

The core part of the algorithm GeneratePDT is the while-loop (lines 9–20 in Fig. 13) which keeps creating PDT nodes using nodes in the candidate tree, and creating new nodes in the candidate tree using the next available Dewey ID in the id lists. We can prove that (Lemma 1 in Appendix A) after loop # t, if a Dewey ID is a result PDT node based on the ids we have processed by the time t, then the Dewey ID must be in the current PDT, CT, or pdt caches of CT. Further, if for any possible completion of the id lists we have processed, this Dewey ID does not satisfy all structural constraints, then it is not in the current PDT, CT, or pdt caches of CT. Hence when the algorithm finishes, both CT and PDT caches are empty, and the result PDT contains and only contains Dewey IDs that satisfy all structural constraints.

7 Experiments

In this section, we show the experimental results of evaluating our proposed techniques developed in the Quark open-source XML database system.

7.1 Experimental setup

In our experiments, we used the 500MB INEX dataset which consists of a large collection of publication records. The excerpt of the INEX DTD relevant to our experiments is shown below.

```
<!ELEMENT books       (journal*)>
<!ELEMENT journal     (title, (sec1|article|sbt)*)>
<!ELEMENT article   (fno, doi?, fm, bdy)>
<!ELEMENT fm        (hdr?, (edinfo|au|kwd|fig)*)>
```

We created a view in which articles (*article* elements) are nested under their authors (*au* elements), and evaluated our system using this view.

The default view definition is shown in Fig. 17. Note that authors.xml contains all authors extracted from the collection. fn:doc(*) returns all document nodes in the collection. We also assume that there are no duplicate co-author names in the same article, which is the case in the INEX dataset.

When running experiments, we generated the regular path and inverted lists indices implemented in Quark (∼1GB each).

We evaluated the performance of four alternative approaches:

Table 1 Experimental parameters

Parameter	Values (default in bold)
Size of Data($\times 100MB$)	1, 2, 3, 4, **5**
# keywords	1, **2**, 3, 4, 5
Selectivity of keywords	Low(IEEE, Computing), **Medium** (Thomas, Control), High (Moore,Burnett)
# of joins	0, **1**, 2, 3, 4
Join selectivity	**1X**, 0.5X, 0.2X, 0.1X
Level of nestings	1, **2**, 3, 4
# of results(K in top-K)	1, **10**, 20, 40
Avg. Size of View Element	**1X**, 2X, 3X, 4X, 5X

Baseline: materializing the view at query time, and evaluating keyword search queries over views implemented using Quark.
TLC: an implementation of GTP with TermJoin for keyword searches and implemented using Timber [33], and with optimizations of GTP on aggregations.
Efficient: our proposed keyword query processing architecture (Sect. 3.1) developed using Quark, with all optimizations and extensions implemented (Sect. 5).
Proj: techniques of projecting XML documents [28].

We have implemented scoring in **Efficient**. Recall that our score computation (Sect. 4.2.2) produces exactly the same TF-IDF scores as if the view was materialized; hence, we do not evaluate the effectiveness of scoring using precision-recall experiments.

Our experimental setup was characterized by parameters in Table 1. *# of joins* is the number of value joins in the view. *Join selectivity* characterizes how many articles are joined with a given author; the default value 1X corresponds to the entire 500MB data; we decrease the selectivity by replicating subsets of the data collection. *Level of nestings* specifies the number of nestings of FLOWR expressions in the view; for value 1, we remove the value join and only leave the selection predicate; for the default value 2, we associate publications under authors; for the deeper views, we create additional FLOWR expressions by nesting the view with one level shallower under the authors list. The rest of the parameters are self-explanatory. In the experiments, when we varied one parameter, we used the default values for the rest.

The INEX data is not recursive by nature and hence our default view based on INEX does not involve repeated tag names. In order to evaluate the effect of recursive tag names, we created a synthetic dataset using a tool similar to the XML data generator used in Niagara [1] so that we can vary the number of recursions. The generated data essentially contains repeated tags of /a/b/c and the view queries consist of //a//b. Further, when varying the number of recursions, to fix the size and the depth of the documents, we insert irrelevant elements to documents with smaller number of recursions.

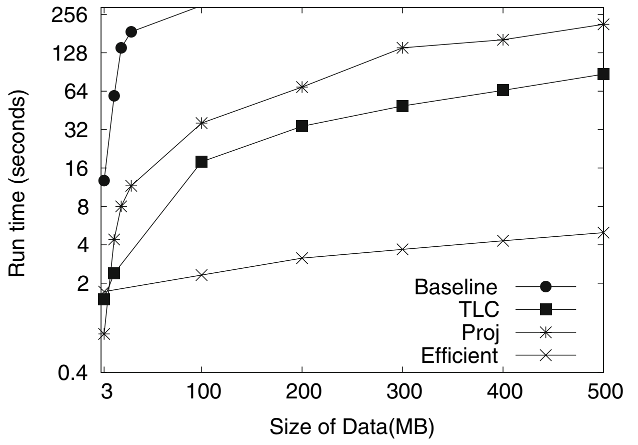


Fig. 18 Varying size of data

All experiments were run on an Intel 3.4Ghz P4 processor running Windows XP with 2GB of main memory. The reported results are the average of five runs with standard deviation less than 10%.

7.2 Performance results

7.2.1 Varying size of data

Figure 18 shows the performance results when varying the size of the data. As shown, it takes EFFICIENT less than 5 seconds to evaluate a keyword query *without* materializing the view over the 500MB data. Second, the run time increases gracefully with the size of the data (note that the y-axis is in *log scale*), because the index I/O cost and the overhead of query processing increases linearly. This indicates that EFFICIENT is a scalable and efficient solution.

In contrast, BASELINE took 59 seconds even for a 13MB data set, which is more than an order of magnitude slower than EFFICIENT. Note the run time includes 58 seconds spent on materializing the view, and 1 second spent on the rest of query evaluation, including tokenizing the view and evaluating the keyword search query.

Further, Fig. 18 shows that EFFICIENT performs $\sim$ ten times faster than TLC. Note that Fig. 18 only shows the time spent by TLC on structural joins and accessing the base data (for obtaining join values); it does not include the time for the remaining query evaluation since they were inefficient and did not scale well (the total running time for TLC, including the time to perform the value join, was more than 5 min on the 100MB data set). TLC is much slower mainly because it relies on (expensive) structural joins to generate the document hierarchy, and because it accesses base data to obtain join values.

Finally, while PROJ merely characterizes the cost of generating projected documents (the cost of query processing and

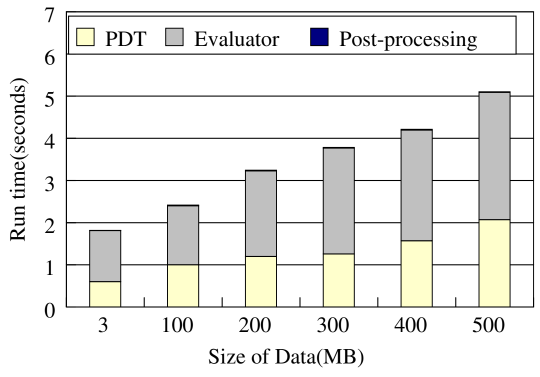


Fig. 19 Cost of Modules

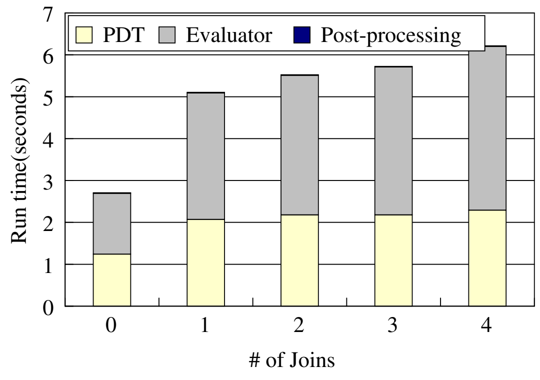


Fig. 21 Varying the number of joins

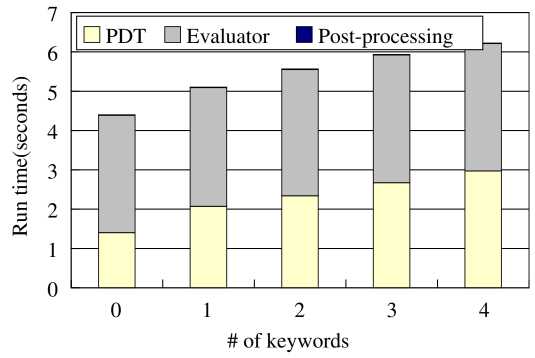


Fig. 20 Varying # keywords

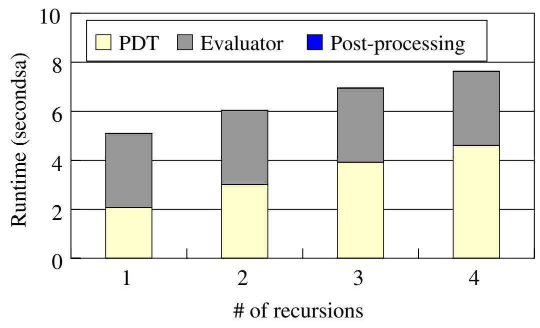


Fig. 22 Varying number of recursions

post-processing are not included), its runtime is ∼15 times slower than EFFICIENT. The main reason is that PROJ scans base documents which leads to relatively poor scalability.

For the rest of the experiments, we focus on EFFICIENT since other alternatives performed significantly slower.

7.2.2 Evaluating overhead of individual modules

Figure 19 breaks down the run time of EFFICIENT and shows the overhead of individual modules—PDT, Evaluator, and Post-processing. As shown, the cost of generating PDTs scales gracefully with the size of the data. Also, the overhead of post-processing, which includes scoring the results and materializing top-K elements, is negligible (which can be barely seen in the graphs). The most important observation is that the cost of the query evaluator dominates the entire cost when the size of the data increases.

7.2.3 Varying other parameters

Varying # of keywords. Figure 20 shows the performance results when varying the number of keywords. The run time for EFFICIENT increases slightly because it accesses more inverted lists to retrieve tf values.

Varying # of joins. Figure 21 shows the performance results when varying the number of value joins in the view definition. As shown, the run time increases with the number of joins mainly because the cost of the query evaluation increases. The run time increases most significantly when the number of joins increases from 0 to 1 for two reasons. First, the case of 0 joins only requires generating a single PDT while the other requires two. More importantly, the cost of evaluating a selection predicate (in the case of 0 joins) is much cheaper than evaluating value joins.

Varying # of Recursions. Figure 22 shows the performance results when varying the number of recursions in the documents. This experiment shows that the run time increases linearly with the number of recursions, while the overhead of the query evaluator grows relatively faster than other modules.

Other results. We also varied the size of the view element, the selectivity of keywords, the selectivity of joins, the level of nestings, and the number of results; the performance results (available in [38]) show that our approach is efficient and scalable with increased size of elements. Finally, the size of PDTs generated with respect to the entire data collection (500MB) is about 2MB, which indicates that our pruning techniques are effective.

8 Related work

There has been a large body of work in the information retrieval community on scoring and indexing [23,24,35,40]. However, they make the assumption that the documents being searched are materialized. In this paper, we build upon existing scoring and indexing techniques and extend them for virtual views. There has also been some recent interest on context-sensitive search and ranking [7], where the goal is to restrict the document collection being searched at run-time, and then evaluate and score results based on the restricted collection. In our terminology, this translates to ranked keyword search over simple selection views (e.g., restricting searches to books with year $>$ 1995). However, these techniques do not support more sophisticated views based on operations such as nested expressions and joins, which are crucial for defining even simple nested views (as in our running example). Supporting such complex operations requires a more careful analysis of the view query and introduces new challenges with respect to index usage and scoring, which are the main focus of this paper.

In the database community, there has been a large body of work on answering queries over views (e.g., [9,19,37]), but these approaches do not support (ranked) keyword search queries. There has also been a lot of recent interest on ranked query operators, such as ranked join and aggregation operators for producing top-k results (e.g., [11,25,34]), where the focus is on evaluating complex queries over ranked inputs. Our work is complementary to this work in the sense that we focus on *identifying* the ranked inputs for a given query (using PDTs). There are, however, new challenges when applying these techniques in our context and we refer the reader to the conclusion for details.

GTP [13] with TermJoin [2] were originally designed to integrate structure and keyword search queries. Since it is a general solution, it can also be applied to the problem of keyword search over views. However, there are two key aspects that make GTP with TermJoin less efficient in our context. First, GTP and TermJoin use relatively expensive structural joins to reconstruct the document hierarchy. Second, GTP requires accessing the base data to support value joins, which is again relatively inefficient. In contrast, our approach uses path indices to efficiently create the PDTs and retrieve join values, which leads to an order of magnitude improvement in performance (Sect. 7).

Finally, our PDT generation technique is related to the technique for projecting XML documents [28]. The main difference is that we use indices to generate PDTs, which leads to a more than tenfold improvement in performance. We refer the reader to Sect. 4 for other technical differences between the two approaches. Our technique is also related to the projection operator in Timber [26] and lazy XSLT transformation of XML documents [36], which, like PROJ, also access the base data for projection.

9 Conclusion and future work

We have presented and evaluated a general technique for evaluating keyword search queries over views. Our experiments using the INEX data set show that the proposed technique is efficient over a wide range of parameters.

There are several opportunities for future work. First, instead of using the regular query evaluator, we could use the techniques proposed for ranked query evaluation (e.g., [11,18,25]) to further improve the performance of our system. There are, however, new challenges that arise in our context because XQuery views may contain non-monotonic operators such as group-by. For example, when calculating the scores of our example view results, extra review elements may increase both the tf values and the document length, and hence the overall score may increase or decrease (non-monotonic). Hence existing optimization techniques based on monotonicity are not directly applicable. Second, our proposed PDT algorithms may be applied to optimize *regular* queries because the algorithms efficiently generate the relevant pruned data, and only materialize the final results.

Acknowledgments We thank Sihem Amer-Yahia at Yahoo! Research for her insightful comments on the draft of the paper. This work was partially funded by NSF CAREER Award IIS-0237644.

Appendix A: Proofs of GeneratePDT

In this section, we formally prove the correctness of the generalized algorithm GeneratePDT. We refer the reader to [38] for the proof of correctness of GenerateQPT.

Specifically, we show that given a QPT, the algorithm GeneratePDT generates the correct PDT that conforms to our PDT definition. Theorem 2 formally describes the correctness of GeneratePDT.

We first introduce some notations. Given a QPT Q, a database D, a node $d \in Nodes(D)$, an environment $\delta \in UE(D, Q)$, we use (d.PathIndex) and *d.InvIndex* to denote the path indices and inverted indices associated with $T(d)$, respectively. Given a QPT Node qn, d.PathIndex.LookUp(qn) returns an ordered list of node ids that correspond to the root to leaf path leading to qn in Q. Each node in the list also satisfies the predicates associated with qn. Given a keyword k, d.InvIndex returns a list of node ids that contains the keyword, along with the tf value.

The following Theorem 2 shows the correctness of the algorithm GeneratePDT:

Theorem 2 *Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, GeneratePDT (Q, $\delta(Q.root)$.PathIndex, $\delta(Q.root)$.InvIndex, KW) = PDT(Q, KW, δ).*

A.1 Notations

We now introduce more notations before proving Theorem 2.

A.1.1 Prefixes

Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, $\delta(Q.root) = d$, *(strLists, invLists) =PrepareList(Q, d.PathIndex, d.InvIndex, KW)*. At a given time # t, we say $H(t, strLists) = \{id \in l | l \in strLists \wedge id \textit{ is retrieved by the time } \# t\}$ is the set of ids that has been retrieved from $strLists$ by the time # t (including t). In our algorithm, t corresponds to the number of loops (lines 9–20 in Fig. 13).

Next, given a QPT node q in Q, for all q' in ancestor nodes of q, and a Dewey ID did in $strLists$ corresponding to q, we use $Prefix(q, did, q')$ to denote the set of prefixes of did that corresponds to q'. Note $Prefix(q, did, q')$ is a set because when the path containing q and q' have the axis '//', there can be multiple matchings of q' in prefixes of did.

Further, $\forall l \in$ L, we say Prefix(l) = {x ∈ Prefix(l.QNode, lid, q) | lid ∈ l, q ∈ anc(l.QNode)} is the set of prefixes of ids in l w.r.t l.QNode, and Prefix(L) = {x ∈ Prefix(l) | l ∈ L}. is the set of prefixes of ids in H(t, strLists).

A.1.2 Pruned document tree based on ID lists

Note since strLists is retrieved by d.PathIndex, ids in strLists can be used to re-create a pruned document tree of T(d). We call lists of ids that can be used to create a valid XML document tree the document-compatible id lists. Essentially in such lists, if two Dewey IDs are identical, then their corresponding path must have the same tag names at each step. If UL is the universe of ordered document-compatible id lists, we use $Comp(H(t, strLists)) \in 2^{UL}$ to denote the universe of completions of id lists in $H(t, strLists)$.

For a set of id lasts $L \in UL$, we use $T(L)$ = (V, E, Tag, Value, Cont) to denote the document tree that contains all and only ids in L. More formally, if rootId(L) is the first id component that all ids in L sares and root(T) is the root node of tree T, then T first satisfies the following properties concerning ids:

- id(root(T(L))) = rootId(L) (The id of the root node is the first component of the Dewey ID in the lists).
- $\forall m, n \in T(L)$, parent(m,n) ⇔ (m.id, n.id ∈ Prefix(L) ∧ parent(id(m), id(n))) (the parent child relations of nodes in T is decided by the Dewey IDs are in the lists).
- $\forall pid \in$ Prefix(L), ∃ n ∈ T, id(n) ∈ T.Cont ∧ id(n) = pid $\wedge \nexists m \in T, m \neq n \wedge$ id(m) = pid. (there is a unique nodes corresponding to each component of the Dewey ID).

Intuitively, T and L has one-to-one mappings on ids. For an ID $did \in$ Prefix(L), if Node(T, did) is the node in T s.t. Node(T, did).id = did, then T further satisfies the following properties:

- $\forall l \in$ L, $\forall id \in$ l, $\forall aq \in$ anc(l.QNode), $\forall pid \in$ Prefix(l.QNode, id, aq), Tag(Node(T, pid)) = aq.name.
- $\forall pid \in$ prefix(L), Value(pid) $\neq null \Rightarrow$ Value (Node(pid)) = Value(pid) $\wedge \forall pid \in$ prefix(L), $id(n)=pid$ ⇒ Value(pid) = null ⇒ Value(Node(T, pid)) = null.

Hence T(H(t, strLists)) denotes the subtree of T(strLists) that contains ids in $H(t, strLists)$.

Further, we use *CT(t)* and *GenPDT(t)* to denote the candidate tree and the PDT the algorithm generates after the loop # t. We also use $CT(t--)$ denote the candidate tree $CT(t-1)$ with new IDs added in the beginning of the loop # t by lines 11–15. and use $CT(t-)$ to denote the candidate tree after we process nodes in the $CT(t-)$ (lines 2–17). We define $C(0-) = C(0--)$ = C(0).

For notational convenience, given a Dewey ID did, if there exists a node $n \in$ *CT(t).V (or GenPDT(t).V, or PDT)*, n.id=did, then we say $did \in$ *CT(t) (or GenPDT(t), or PDT)*. And given a id pid, a QPT Q, a set of keywords KW, L ∈ UL, we say the predicate Qualified(pid, Q, KW, L) = true ⇔ pid ∈ PDT(Q, KW, {Q.root ⇒ T(L).root}).

A.2 Proofs

At a high level, the algorithm GeneratePDT consists of three steps. First, it invokes PrepareList to construct lists of Dewey ids, ordered by id, that correspond to nodes without mandatory children nodes in the QPT. Then, it initializes the candidate tree using the minimum ID from each id list. Next, it enters a loop which keeps creating PDT nodes using qualified (defined later) CT nodes and creating new CT nodes using available IDs. The algorithm terminates after processing all IDs, and removing all nodes in the CT.

The core part of the algorithm GeneratePDT is the while-loop (lines 9–20 in Fig. 13) which keeps creating PDT nodes using nodes in the candidate tree, and creating new nodes in the candidate tree using the next available id in the id lists. We first prove a theorem that characterizes the invariant of this loop.

Lemma 1 *Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after the loop # t,*

(a) $\forall pid \in$ *Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists)) = true* $\Rightarrow$ *(pid* $\in GenPDT(t) \vee pid \in CT(t)) \vee \exists n \in CT(t).V$, *pid* $\in$ *n.PDTCache) (qualified nodes are in the candidate tree or the result PDT), and*

(b) $\forall id \in$ *GenPDT(t), id* $\in$ *Prefix(H(t, strLists))* $\wedge$ *Qualified(pid, Q, KW, H(t, strLists)) = true. (all nodes in the PDT are qualified)*

Lemma 1 indicates that after the loop # t, if a Dewey ID is a result PDT node based on the ids we have processed by t, then the id must be kept in GenPDT(t), CT(t), or pdt caches of CT(t). Further, if for any possible completion of the id lists we have processed, this Dewey ID is not qualified, then it is not in CT(t), GenPDT(t), or pdt caches of CT(t).

A.2.1 Supporting lemmas for Lemma 1

We now present a set of lemmas that will be used in the proof of Lemma 1. Proofs of these supporting lemmas are available at Appendix B.

First, by the definition of PDT, it is easy to show the following lemma:

Lemma 2 (Monotonicity) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then for any loop # t,*

(a) $\forall pid \in$ *Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists)) = true* $\Rightarrow \forall L \in Comp(H(t, strLists))$, *Qualified(pid, Q, KW, L) = true.*

(b) $\forall cn \in CT(t)$, $\forall cnq \in$ *cn.CTQNodeSet, id(cn)* $\in$ *CE (cnq. T(H(t, strLists)).root)* $\Rightarrow \forall t' \geq t$, *(cn* $\in$ *CT(t')* $\Rightarrow$ *id(cn)* $\in$ *CE(cnq, T(H(t', strLists)).root).*

(c) $\forall cn \in CT(t)$, $\forall cnq \in$ *cn.CTQNodeSet, id(cn)* $\in$ *PE (cnq. T(H(t, strLists)).root)* $\Rightarrow \forall t' \geq t$, *cn* $\in$ *CT(t')* $\Rightarrow$ $\in$ *PE(cnq. T(H(t', strLists)).root).*

The key idea is that the membership of a PDT node is determined by existence of its ancestor nodes and its mandatory children nodes in the PDT. Hence given a QPT and a set of IDs SI, if an ID is included in the PDT as per the definition, then this id is also included in the PDT using any superset of SI because all of its ancestor and children nodes must also be in the superset.

Given a QPT q and a node $qn \in q$, we say MC(qn) = {qnc | (qn, qnc, axis, 'm') ∈ q.E ∧ axis = '/' or '//'} is the mandatory children nodes of qn in q. For each edge e in the QPT, we represent e using a 4-tuple (parent, child, axis, ann) where parent and child are the parent and child node of e, respectively, axis is '/' or '//', and ann is 'o' or 'm'.

Given a CT node cn, a QPT node qn ∈ MC(CT. CTQNode), the following Lemma 3 indicates that the value of cn.DM [qcd] corresponds to whether cn has a child/descendant node that is also a candidate element. Since we add new ids by calling AddNewCTNodes(), we use $List_t$ to denote the lists of IDs that have been retrieved after calling t times of AddNewCTNodes, and CT_t denote the candidate tree after calling t times of AddNewCTNodes.

Lemma 3 (DescendantMap) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after adding # t IDs,*
$\forall cn \in CT_t$, $\forall cnq \in$ *cn.CTQNodeSet* $\forall qcd \in$ *MC (cnq.QNode), cnq.DM[qcd] = 1* $\Leftrightarrow$
(((cnq.QNode, qcd, '/', 'm') $\in$ *Q.E* $\Rightarrow \exists l \in List_t$, $\exists lid \in l$, $\exists cid \in$ *Prefix(l.QNode, lid, qcd),* $\exists ce \in$ *CE(qcd,* $T(List_t)$*.root), ce.id = cid* $\wedge$ *id(cn)* $\in$ *Prefix(l.QNode, lid, cnq.QNode)* $\wedge$ *parent(id(cn), cid))* $\wedge$
((cnq.QNode, qcd, '//', 'm') $\in$ *Q.E* $\Rightarrow \exists l \in List_t$, $\exists lid \in l$, $\exists cid \in$ *Prefix(l.QNode, lid, qcd),* $\exists ce \in$ *CE(qcd,* $T(List_t)$*.root), ce.id = cid* $\wedge$ *id(cn)* $\in$ *Prefix(l.QNode, lid, cnq.QNode)* $\wedge$ *anc(id(cn), cid)))*

Since at each loop (lines 9–20), we start by adding new IDs corresponding to the current left most path, Then it is easy to infer the following lemma from Lemma 3:

Lemma 4 (DM) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then for every loop #t,*
$\forall cn \in CT(t - -)$, $\forall cnq \in$ *cn.CTQNodeSet* $\forall qcd \in$ *MC (cnq.QNode), cnq.DM[qcd] = 1* $\Leftrightarrow$
(((cnq.QNode, qcd, '/', 'm') $\in$ *Q.E* $\Rightarrow \exists l \in$ *H(t, strLists),* $\exists lid \in l$, $\exists cid \in$ *Prefix(l.QNode, lid, qcd),* $\exists ce \in$ *CE(qcd, T(H(t, strLists)).root), ce.id = cid* $\wedge$ *id(cn)* $\in$ *Prefix(l.QNode, lid, cnq.QNode)* $\wedge$ *parent(id(cn), cid))* $\wedge$
((cnq.QNode, qcd, '//', 'm') $\in$ *Q.E* $\Rightarrow \exists l \in$ *H(t, strLists),* $\exists lid \in l$, $\exists cid \in$ *Prefix(l.QNode, lid, qcd),* $\exists ce \in$ *CE(qcd, T(H(t, strLists)).root), ce.id = cid* $\wedge$ *id(cn)* $\in$ *Prefix(l.QNode, lid, cnq.QNode)* $\wedge$ *anc(id(cn), cid)))*

Now, Lemma 5 indicates that if the flag InPdt of a CT node is true, the id of this node is qualified.

Lemma 5 (InPdt) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList (Q, d.PathIndex, d.InvIndex, KW), then at the loop # t,*

(a) $\forall n \in CT(t-)$, $\forall nq \in$ *n.CTQNodeSet, nq.InPdt = true* $\Rightarrow$ *cn* $\in$ *PE(nq.QNode, T(H(t, strLists)).root).*

(b) $\forall n \in CT(t-)$.*LeftMostPath*, $\forall nq \in$ *n.CTQNodeSet*, $t > 0 \wedge cn \in$ *PE(nq.QNode, T(H(t, strLists)).root))* $\Rightarrow$ *nq.InPdt* = *true* $\wedge$ ($\forall t' \geq t$, $n \in$ *CT(t')* $\Rightarrow$ *nq.InPdt* = *true* $\wedge n \in CT(t'-) \Rightarrow$ (*nq* $\in$ *n.CTQNodeSet* $\wedge$ *nq.InPdt* = *true*) $\wedge n \in CT(t'--) \Rightarrow$ (*nq* $\in$ *n.CTQNodeSet* $\wedge$ *nq.InPdt* = *true*)).

The following Lemma 6 characterizes the properties of pdt cache. Note that for ease of exposition, we additionally associate each node in the pdt cache with a set of QPT node, denoted as PDTQNodes, as CTQNodeSet in CT nodes. Formally, we change line 10 in Fig. 15 to the following:

pdtCacheNode.PDTQNodes.add(q.QNode, q.PL)

Then for a node *n* in the pdt cache, it is easy to see that PL(n) = {x ∈ q.PL| q ∈ n.PDTQNodes}, and we use n.PL and PL(n) interchangeably.

Lemma 6 (PDTCache) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList (Q, d.PathIndex, d.InvIndex, KW), then at the loop # t,*

(a) $\forall cn \in$ *CT(t)*, $\forall cnp \in$ *cn.pdtCache*, $\forall q \in$ *cnp. PDTQNodes*, $\exists ce \in$ *CE(q, T(H(t, strLists)).root), ce.id = cnp.id (nodes in the pdt caches satisfy the descendant restrictions).*

(b) $\forall cn \in$ *CT(t)*, $\forall cnp \in$ *cn.pdtCache*, (*PL(cnp)* $\neq \emptyset \wedge$ $\forall cnpp \in$ *PL(cn), cnpp.InPdt = false*) $\Rightarrow$ *Qualified (cnp.id, H(t, strLists)) = false (if parents are not qualified, then the node itself is not qualified).*

(c) $\forall cn \in$ *CT(t)*, $\forall cnp \in$ *cn.pdtCache*, (*PL(cnp)* $= \emptyset \vee$ $\exists cnpp \in$ *PL(cnp), cnpp.InPdt = true*) $\Rightarrow$ *Qualified (cnpp.id, H(t, strLists)) = true (if the node does not have parents or at least one parent is qualified, then the node is qualified).*

For notational convenience, given a Dewey ID did and a candidate tree CT, if there exists a node n ∈ CT and did ∈ n.pdtCache, then we say did ∈ pdtCache(CT).

Lemma 7 (Completeness of CT) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) = d$ *and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then at the loop # t,* $\forall pid \in$ *Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists))=false* $\wedge \exists L \in$ *Comp(H(t, strLists)), Qualified(pid, Q, KW, L)=true* $\Rightarrow$ *pid* $\in$ *CT(t)* $\vee$ *pid* $\in$ *pdtCache(CT(t)).*

Lemma 7 indicates that if a Dewey ID could potentially be a qualified ID, then it will be included in the candidate tree.

Finally, when the algorithm initializes the candidate tree (lines 6–7 in Fig. 13), it simply creates nodes in the candidate tree using the minimum ids from each list, and does not remove nodes or create node in the pdt cache. Therefore if *MinimumID(l)* is the minimum Dewey ID in the list *l*, then it is straightforward to infer the following lemma:

Lemma 8 (Initialization of CT) *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root)$*=d and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after initializing the candidate tree CT,*

(a) $\forall id \in CT$, $\exists l \in$ *strLists*, $\exists q \in$ *anc(l.QNode)*, $\exists pid \in$ *Prefix(l.QNode, MinimumID(l), q), id = pid*

(b) $\forall l \in$ *strLists*, $\forall q \in$ *anc(l.QNode)*, $\forall pid \in$ *Prefix (l.QNode, MinimumID(l), q), pid* $\in CT$.

A.2.2 Proofs of Lemma 1

We separate Lemma 1 into two parts and prove each of them separately.

Lemma 9 *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) =$ *d and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after the loop # t,* $\forall pid \in$ *Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists))=true* $\Rightarrow (pid \in GenPDT(t) \vee pid \in CT(t)) \vee \exists n \in CT(t).V$, *pid* $\in$ *n.PDTCache (qualified nodes are in the candidate tree or the PDT).*

Lemma 10 *Given a set of keywords KW, a QPT Q, an XML database D, an environment* $\delta \in UE(D, Q)$, *if* $\delta(Q.root) =$ *d and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after the loop # t,* $\forall id \in$ *GenPDT(t), id* $\in$ *Prefix(H(t, strLists))* $\wedge$ *Qualified(id, Q, KW, H (t, strLists)) = true. (all nodes in the PDT are qualified).*

We first prove Lemma 9.

Proof We prove Lemma 9 by induction on the loop # t.

Base case: t = 0. In this case, the algorithm just initializes the candidate tree using the minimum ids from each list in strLists, and it is easy to see that GenPDT(t) = null, and $\forall n \in CT(t)$, n.PDTCache = null. On the other hand, by Lemma 8, we know that $\forall l \in$ strLists, $\forall q \in$ anc(l.QNode), $\forall pid \in$ Prefix(l.QNode, MinimumID(l), q), $pid \in CT$. This implies that $\forall pid \in$ Prefix(H(0, strLists)), pid ∈ CT(0) and hence Lemma 9 is vacuously tree.

Induction Hypothesis: Suppose the lemma holds for loop # n, and we need to show it also holds for loop # n+1.

Given a list l, if Q(t, l)={x ∈ Prefix(l.QNode, id, q) | q ∈ anc(l.QNode) ∧ id ∈ l ∧ Qualified(x, Q, KW, H(t, strLists)) = true} is the set of qualified ids in *l* at a given loop # t, and

Q(t) = {x ∈ Q(t, l) | l ∈ H(t, strLists)} is the set of all qualified ids at the loop # t, we prove the lemma in three different cases, one for each different case of *id* ∈ Q(n+1). (a) *id* ∈ Q(n), i.e., id is already qualified at the loop # n; (b) *id* ∈ *Prefix(H(n, strLists))* ∧ *id* ∉ *Q(n)*, i.e., id is in Prefix(H(n, strLists)) and just becomes qualified at the loop # n+1; and (c) *id* ∈ *Prefix(H(n+1, strLists))- Prefix(H(n, strLists))*, i.e., id is just introduced at the loop # n+1.

Case a: id ∈ Q(n). In this case, *id* is already qualified at the loop # n. Therefore by I.H., *id* ∈ CT(n), *id* ∈ GenPDT(n), or ∃*cn* ∈ CT(n), *id* ∈ cn. pdtCache. Now we discuss these three cases separately.

Case a.1. First, if $id \in GenPDT(n)$, then by the algorithm GenPDT(n) ⊆ GenPDT(n+1), we know that *id* ∈ GenPDT(n+1).

Case a.2. Second, if *id* ∈ CT(n), there are further two different mini-cases.

Case a.2.1. First, if $id \notin CT((n+1)--)$.LeftMostPath, then by the algorithm, we know that *id* will not be processed at the loop # n+1, and hence *id* ∈ CT(n+1).

Case a.2.2. Second, if $id \in CT((n+1)--)$. LeftMostPath, assume *cn* is the node in $CT((n+1)--)$. LeftMostPath s.t. id(cn)=id. Since Qualified(id, Q, KW, H (n, strLists)) = true, by definition we know that ∃*cnq* ∈ *cn.CTQNodeSet*, ∀*cnc* ∈ MC(cnq), ((cnq.QNode, cnc, '/', 'm') ∈ Q.E ⇒ ∃*ce* ∈ CE(cnc, T(H(n, strLists)). root), parent(id(cn), ce.id)) ∧ ((cnq.QNode, cnc, '//', 'm') ∈ Q.E ⇒ ∃*ce* ∈ CE(cnc, T(H(n, strLists)). root), anc(id(cn), ce.id)) (*). Hence at the loop n+1, by Lemma 3, we know that ∀*qcd* ∈ MC(cnq), cnq.DM [qcd] = 1.

Further, also by Qualified(id, Q, KW, H(n,strLists))=true, we know that ∃*cqn* ∈ cn.CTQNodes s.t. cqn satisfies the property (*) as described above and (cnq.PL = ∅ ∨ ∃*cnp* ∈ $CT((n+1)--)$. LeftMostPath, ∃*p* ∈ cnp.CTQNodeSet, p ∈ cnq.PL ∧ cnp ∈ PE(p, T(H(n, strLists)).root).

Then by Lemma 5, at the loop n+1, p.InPdt = true. Hence by lines 2–5, *id* ∈ GenPDT(n+1).

Case a.3. Third, if ∃*cn* ∈ CT(n).LeftMostPath, *id* ∈ cn.pdtCache. If *cn* ∈ CT(n+1), then by the algorithm the nodes in cn.pdtCache will not be removed, and hence the lemma holds. Otherwise we can use the same argument as in Case a.2.2 and show that cn.id ∈ GenPDT(n+1).

Case (b): id ∈ H(n, strLists) ∧ id ∉ Q(n). First, since *id* ∈ Q(n+1), by Lemma 7, we know that *id* ∈ CT(n) ∨ ∃*cn* ∈ CT(n), *id* ∈ cn.pdtCache. Then we can use the similar argument to reason $CT((n+1)--)$, as in Case a.2 and a.3, and show the lemma holds.

Case (c): id ∈ Prefix(H(n+1, strLists))– H(n, strLists). In this case, the algorithm will first add *id* in CT(n) and then process $CT((n+1)--)$. LeftMostPath. Then if *id* ∉ CT(n). LeftMostPath, the lemma is vacuously true; otherwise we can prove the lemma using the same argument as in Case a.2 and Case a.3.

Therefore the lemma holds for all ids in Q(n+1). □

We now prove Lemma 10.

Proof We prove the lemma by induction on the loop # t.

Base case: t = 0. It is vacuously true because GenPDT(t) = null.

Induction Hypothesis : Assume the lemma holds for the loop # t ≤ *n*, we show that it also holds for loop # n+1.

First, ∀*id* ∈ GenPDT(n) ∩ GenPDT(n+1), by I.H., we know that ∃*id* ∈ Prefix(H(n, strLists)) ∧ Qualified(id, Q, KW, H(n, strLists)) = true.

Therefore by Lemma 2, we know Qualified(id, Q, KW, H(n+1, strLists)) = true, and hence the lemma holds.

Now we prove the lemma for all *g* ∈ (GenPDT(n+1) − GenPDT(n)). By the algorithm there are three possible cases, one for each different scenario where *g* is created in GenPDT(n+1).

Case 1: g.id $\in CT((n+1)--)$. In this case, since g is in GenPDT(n+1), by line 5 we know that ∃*q* ∈ g.CTQNodes, q.InPdt = true, and hence by Lemma 5, Qualified(g.id, Q, KW, H(n+1, strLists)) = true.

Case 2: ∃*cn* $\in CT((n+1)--)$, *g.id ∈ cn. PDTCache.* Since g is created in GenPDT(n+1), by line 21 in Fig. 15, we know that either (1) PL(g) = ∅ or (2) ∃*p* ∈ PL(g), p.inPDT = true. Hence by Lemma 6, we know that Qualified(g.id, Q, KW, H(n+1, strLists)) = true.

Case 3: g.id∈Prefix(H(n+1, strLists)) - Prefix(H(n, strLists)). In this case, g.id $\in CT((n+1)--)$, and hence we can use the similar argument to Case 1 to show the lemma holds. □

Appendix B: Proofs of supporting lemmas

Proof of Lemma 3

Proof First, if MC(cnq) = ∅, then the lemma is vacuously true and hence we only consider the case where MC(cnq) ≠ ∅.

"⇒"
We prove the inductions on # t.

Base case: t = 0. In this case, CT(0) is empty and thus the lemma is vacuously true.

Induction Hypothesis: Assume the lemma holds for t≤n, we show that it also holds for t = n+1.

Note by the algorithm, n+1 and n can be in the same or different loops in lines 9–20. However, since we never modify the value of DM after adding IDs, we do not differentiate these two cases.

Now we assume that after adding the ID at the time n+1, given a $cn \in CT_{n+1}$, $cnq \in$ cn.CTQNodeSet, $qcd \in$ MC(cnq), cnq.DM[qcd] = 1. There are four different cases to consider. (a) cn $\in CT_n \wedge$ cnq $\in CT_n \wedge$ cnq[qcd] = 1 at the time n, and (b)cn $\in CT_n \wedge$ cnq $\in CT_n \wedge$ cnq[qcd] = 0 at the time n, and (c)cn $\in CT_n \wedge$ cnq $\notin CT_n$, and (d)cn $\notin CT_n$.

Case (a). In this case, by I.H., we know that ((cnq.QNode, qcd, '/', 'm') ∈ Q.E ⇒ ∃l ∈ $List_n$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_n$).root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ parent(id(cn), cid))∧
((cnq.QNode, qcd, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_n$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_n$).root), ce.id = cid ∧ id(cn) ∈ Prefix (l.QNode, lid, cnq.QNode) ∧ anc(id(cn), cid)).

Hence by the definition of candidate elements, it is easy to infer that ((cnq.QNode, qcd, '/', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_{n+1}$).root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ parent(id(cn), cid)) ∧
((cnq.QNode, qcd, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T ($List_{n+1}$).root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ anc(id(cn), cid)).

Hence the lemma holds.

Case (b). In this case, we show the lemma by induction on the depth of qcd.

Base case: qcd is the leaf node. In this case, since cnq. DM[qcd] is set to 1, if nid∈ l the id we add at the time n+1, then we can infer that $\exists qid$ ∈ Prefix(l.QNode, nid, qcd). Further, since qcd is the leaf node, by definition of candidate elements and by the specification of path index, we know that $\exists qid$ ∈ CE(qcd, T($List_{n+1}$).root). Further since we set cnq.DM[qcd] = 1, we know that cqn ∈ qcd.PL, therefore we can finally conclude that (cnq.QNode, qcd, '/', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_{n+1}$).root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ parent(id(cn), cid)) ∧
((cnq.QNode, qcd, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T ($List_{n+1}$).root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ anc(id(cn), cid)).

Induction Hypothesis: Assume the lemma holds for qcd of depth ≥ d, we need to show the lemma also holds for d-1.

If MC(qcd) = ∅, then we can use the similar argument as the base case and show the lemma holds. Otherwise MC(qcd) ≠ ∅.

There are two mini-cases here, depending on whether there exists a child node of cn in CT_n which contains qcd in its CTQNodeSet.

First, assume $\exists qn \in CT_n$, $\exists qc$ ∈ cn.CTQNodeSet, qcd = qc.QNode ∧ $\forall mq$ ∈ MC(qc), qc.DM[mq] = 1 at the time n+1. In this case, since at the time n, cnq.DM [qcd] = 0, intuitively we know that certain descendant restrictions of qcd are not satisfied at the time n. If X = {x|x∈MC(qcd) $wedge$ qc.DM[x] = 1 in CT_n}, and Y = {y|x∈MC(qcd) ∧ qc.DM[y] = 0 in CT_{n+1}}.

Then by I.H. on the number n, we know at the time n, the lemma holds for all x in X. Further, by I.H. on the depth, we know the lemma also holds for all y in Y.

Therefore we know that at the time n+1, $\forall mq$ ∈ MC(qc), (qcd, mq, '/', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, mq), $\exists ce$ ∈ CE(mq, T($List_{n+1}$) .root), ce.id = cid ∧ id(qn) ∈ Prefix(l.QNode, lid, qcd) ∧ parent(id(qn), cid)) ∧
((cnq.QNode, qcd, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, mq), $\exists ce$ ∈ CE(mq, T($List_{n+1}$) .root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, qcd) ∧ anc(id(cn), cid)).

Therefore qn ∈ CE(qcd, T($List_{n+1}$).root), and hence (cnq.QNode, qcd, '/', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_{n+1}$) .root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ parent(id(cn), cid)) ∧
((cnq.QNode, qcd, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_{n+1}$, ∃lid ∈ l, ∃cid ∈ Prefix(l.QNode, lid, qcd), $\exists ce$ ∈ CE(qcd, T($List_{n+1}$) .root), ce.id = cid ∧ id(cn) ∈ Prefix(l.QNode, lid, cnq.QNode) ∧ anc(id(cn), cid)).

Second, if $\nexists qn \in CT_n$, $\exists qc$ ∈ cn.CTQNodeSet, qcd = qc.QNode. In this case, since we only add a single Dewey ID, we can use the similar induction as in the first case from bottom up and show the lemma holds.

Case (c) and (d) In this case, we just add the QPT node cqn at the time n+1. Then we can also show the lemma using an easy induction on the depth of qcd, similar to Case (b).

"⇐"
We prove the inductions on # t.

Base case: $t = 0$. In this case, no IDs have been retrieved and CT(0) is empty and hence the lemma is vacuously true.

Induction Hypothesis: Assume the lemma holds for t≤n, we show that it also holds for t = n+1.

If B denote the RHS of the statement, then given a cn ∈ CT_{n+1}, cnq ∈ cn.CTQNode, qcd ∈ MC(cqn.QNode), there are five cases to consider. (a) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∈ CT_n ∧ B =true, and (b) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∈ CT_n ∧ B = false, and (c) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∉ CT_n ∧ B = false, and (d) cn ∈ CT_n ∧ cnq ∉ CT_n ∧ qcd ∉ CT_n ∧ B = false, and (e) cn ∉ CT_n ∧ cnq ∉ CT_n ∧ qcd ∉ CT_n ∧ B = false.

We now show each of them separately.

Case (a). In this case, by I.H., we know that cnq. DM[qcd] = 1. By the algorithm, we never change the value from 1 to 0, and hence the lemma holds.

Case (b). In this case, we can infer that cid ∉ CE(qcd, T($List_n$) .root. But since we assume that cid ∈ CE(qcd, T($List_{n+1}$).root, we know that MC(qcd) $\neq \emptyset$, and $\exists mq$ ∈ MC(qcd), (qcd, mq, '/', 'm') ∈ Q.E, ⇒ ∄l ∈ $List_n$, ∃lid ∈ l, ∃mid ∈ Prefix(l.QNode, lid, mq), mid ∈ CE(mq, T($List_n$) .root), ∧ cid ∈ Prefix(l.QNode, lid, qcd) ∧ parent(cid, mid)) ∧
((qcd, mq, '//', 'm') ∈ Q.E ⇒ ∃l ∈ $List_n$, ∃lid ∈ l, ∃mid ∈ Prefix(l.QNode, lid, mq), mid ∈ CE(mq, T($List_n$).root) ∧ cid ∈ Prefix(l.QNode, lid, qcd) ∧ anc(cid, mid)) (*).

We assume qn is a node in $CT_n + 1$ and CT_n s.t. qn.id = cid. we say X = {x|x ∈ MC(qcd), property (*) does not hold}, and Y = {y|y ∈ MC(qcd), property (*) holds}.

First, for all x in X, by I.H., we know that the lemma holds. Hence $\forall x \in X$, if qnc ∈ qn.CTQNode and qnc.QNode = qcd, then qnc.DM[x] = 1. For all y in Y, we can show that qnc.DM[y] = 1 in CT_{n+1} by induction on the depth of y. If y is the leaf node and cy is the CT node corresponding to y, then by the algorithm we will set qnc.DM[y] to be 1. Inductively, if y is the non-leaf node. Then if MC(y) = ∅, by the algorithm, we will also set qnc.DM[y] = 1. Otherwise by I.H. on the depth, we know for all yy ∈ MC(y), the corresponding entries in DM are set to 1, and hence qnc.DM[y] is set to 1. Hence by the algorithm, cnq.DM[qcd] is set to 1.

Hence the lemma holds in this case.

Case (c), (d), and (e). In all of these cases, we can prove induction on the depth of qcd in a similar fashion to Case (b). As the base case, if qcd is the leaf node, if $did \in l$ is the single Dewey ID that we add to CT_{n+1}, we know that cid ∈ Prefix(l.QNode, did, qcd), and hence by the algorithm, we will set cnq.DM[qcd] to be 1. Inductively, if qcd is a non-leaf node, then if MC(qcd) = ∅, we can show the lemma similar to the base case. Otherwise by I.H. on the depth, if cnn is the CT node s.t. $\exists qn$ ∈ cnn.CTQNodeSet, qn.QNode = qcd, then $\forall x$ ∈ MC(qcd), qn.DM[x] = 1. And hence by the algorithm, we will set cnq.DM[qcd] to be 1. □

Proof of Lemma 5

Proof We first prove (a) by induction on the loop # t.

Base case: $t = 0$. The lemma is vacuously true since in $CT(0-)$, we do not change the values of InPdt from false to true.

Induction hypothesis: Assume the lemma holds for loop # ≤ t, we show it also holds for loop # t+1.

First, if cn ∈ $CT(t-)$ and cnq ∈ cn.CTQNodeSet and cnq.InPdt = true, then by I.H., we know that the lemma holds. Otherwise the value of inPdt is set to true at the loop # t+1.

We show the lemma holds in this case by inductions on the depth of nodes, starting from the root.

Base case: depth = 0. In this case, the node cn is the root node and hence we know that $\forall cnq$ ∈ cn.CTQNode-Set, cnq.QNode is also the root node (in fact, by definition there is only a single node in cn.CTQNodeSet in this case). Hence cnq.PL = ∅, and by line 4 in Fig. 15, cnq.inPdt is set to true when $\forall i$ ∈ cnq.DM[i] = 1. By Lemma 3, this implies that cn ∈ CE(cnq.QNode, T(H(t, strLists)).root). Therefore by definition of PE, we know that cn ∈ PE(cnq.QNode, T(H(t, strLists)).root), and hence the lemma holds.

Induction hypothesis: Assume the lemma holds for nodes of depth ≤ n, we now show the lemma also holds for nodes of depth n+1.

Given cnq ∈ cn.CTQNodes, if cnq.inPdt = true, then by lines 4–5 in Fig. 15, we know that $\forall i$ ∈ cnq.DM[i] = 1. By Lemma 3, this implies that cn ∈ CE(cnq.QNode, T(H(t+1, strLists)).root). We also know that cnq.PL = ∅ or $\exists p$ ∈ cnq.PL, If cnq.PL = ∅, then cnq is the root node in the QPT and by definition, cn ∈ PE(cnq.QNode, T(H(t+1, strLists)).root). If $\exists p$ ∈ cnq.PL, q.inPdt = true, and if cnp is the CT node s.t. p ∈ cnp. CTQNodeSet, then by the algorithm, we know that cnp is an ancestor node of p. Then if cnp.InPdt = true before the loop # t+1, we can apply I.H. on the loop # t and using Lemma 2 to infer that cnp ∈ PE(p, T(H(t+1, strLists)).root); otherwise we can use I.H. on the depth of nodes and infer that cnp ∈ PE(p, T(H(t+1, strLists)).root). Hence by definition of PDT, we know that cnp ∈ PE(p, T(H(t+1, strLists)).root). Hence (a) holds.

(b) We only show that $\forall n \in CT(t-)$.LeftMostPath, $\forall nq$ ∈ n.CTQNodeSet, cn ∈ PE(nq.QNode, T(H(t+1, strLists)) .root)) ⇒ nq.InPdt = true.

It is straightforward to infer that $\forall n \in CT(t-)$. LeftMostPath, $\forall nq \in$ n.CTQNodeSet, cn $\in$ PE(nq.QNode, T(H(t, strLists)).root)) $\Rightarrow$ nq.InPdt = true $\wedge$ ($\forall t' \geq t$, n $\in$ CT(t') $\Rightarrow$ nq.InPdt = true $\wedge$ n $\in CT(t'-)$ $\Rightarrow$ (nq $\in$ n. CTQNodeSet $\wedge$ nq.InPdt = true) $\wedge$ n $\in CT(t'--)$ $\Rightarrow$ (nq $\in$ n.CTQNodeSet $\wedge$ nq.InPdt = true)) because we never change the flag from true to false.

We now prove (b) by induction on the loop # t.

Base case: t = 1. We prove the lemma holds in this case by induction on the depth of the nodes in $CT(1-)$.

Base case: depth = 0. In this case, the node *cn* is the root node and hence we know that $\forall cnq \in$ cn.CTQNodeSet, cnq.QNode is also the root node. In fact, by definition there is only a single node in cn.CTQNodeSet, call it sq. Since cn $\in$ PE(sq.QNode, T(H(t, strLists)).root), we can infer that cn $\in$ CE(sq.QNode, T(H(t, strLists)).root). Hence by Lemma 3, $\forall i \in$ cnq.DM[i] = 1. Further, since sq is the root node in the QPT, sq.PL = $\emptyset$. Then by line 4 in Fig. 15, cnq.inPdt is set to true. Hence (b) holds.

Induction hypothesis: Assume the lemma holds for nodes of depth $\leq$ n, we now show the lemma also holds for nodes of depth n+1.

Given *cnq* $\in$ cn.CTQNodes, if cnq is the root node in the PDT, then we can use the similar argument to the base case and show (b) holds. If cnq is non-root node and assume p is the parent node of cnq. Assume cnp is an ancestor node of cn and *p* $\in$ cnp.CTQNodes, then *cn* $\in$ PE(cnq, T(H(t+1, strLists)).root) implies that cnp $\in$ PE(p, T(H(t+1, strLists)).root). By I.H., we know that cnp.InPdt = true. Further *cn* $\in$ PE(cnq, T(H(t+1, strLists)).root) also implies that *cn* $\in$ CE(cnq, T(H(t+1, strLists)).root, hence by Lemma 3, $\forall i \in$ cnq.DM[i] = 1. Therefore by lines 4–5 in Fig. 15, cnq.InPdt will be set to true.

Hence (b) holds in the base case.

Induction hypothesis: Assume the lemma holds for loop # $\leq$ t, we now show the lemma also holds for loop # t+1.

Given *cn* $\in CT((t+1)-)$.LeftMostPath, *cnq* $\in$ cn. CTQNodes, if *cn* $\in CT(t-)$ .LeftMostPath and id(cn)$\in$ PE(cnq, T(H(t, strLists)) .strLists)), the by I.H., we know that cnq.InPdt = true at the loop # t. Since we never change it from true to false, the lemma holds.

Otherwise cnq.InPdt is set to true at the loop # t+1. We can use the similar induction on the depth of the nodes as in the base case to show the lemma holds.

Proof of Lemma 6

Proof (a) It is easy to prove (a) by lines 11 in Fig. 15 using Lemma 3.

(b) We prove (b) by considering different cases corresponding to when the node is created in the pdt cache and when the parent list is updated. At the loop # t, given *cn* $\in$ CT(t+1), *x* $\in$ cn.PdtCache, if *x* is just created in cnp.PdtCache, then by definition of PL, we know that if $\forall q \in$ x.PDTQNodes, $\forall p \in$ q.PL, Qualified(id(p), H(t, strLists)) = false implies Qualified(id, H(t, strLists)) = false.

Otherwise x is created at loop x $\leq$ t and is updated. We can show the lemma by inductions on the number of update times. The base case is just shown. Inductively, we assume the (b) holds for the case where PL(x) is updated n times. Now PL(x) is updated again by line 28 in Fig. 15. Assume q is replaced by q.PL, by definition we know that $\forall qp \in$ q.PL, Qualified(id(qp), H(t, strLists)) = false implies Qualified(id(q), H(t, strLists)) = false. Further, we know that by I.H., Qualified(id(q), H(t, strLists)) = false implies that Qualified(x.id, H(t, strLists)) = false. Since we assume $\forall qp \in$ q.PL, Qualified(id(qp), H(t, strLists)) = false, we can conclude that Qualified(x.id, H(t, strLists)) = false.

(c) can also be shown in a similar fashion as in (b).

Proof of Lemma 7

Proof We can prove this lemma by induction on t.

Base case: t = 0. In this case, all ids in H(t, strLists) are in CT(t) and therefore the lemma is vacuously true.

Induction Hypothesis: Assume the lemma holds for $t \leq$ n, we need to show the lemma holds for n+1.

We show an equivalent statement as follows. *pid* $\notin$ CT(n+1) $\wedge$ *pid* $\notin$ pdtCache(CT(n+1)) $\wedge$ Qualified(pid, Q, KW, H(n+1, strLists)) = false $\Rightarrow \nexists L \in$ Comp(H(n+1, strLists)), Qualified(pid, Q, KW, L) = true.

There are two cases to consider depending on whether pid is in Prefix(H(n, strLists)).

Case 1: pid $\in$ Prefix(H(n, strLists)). First, by Lemma 2, Qualified(pid, Q, KW, H(n+1, strLists)) = false implies Qualified(pid, Q, KW, H(n, strLists)) = false. Then we have two different cases to consider.

Case 1.1: pid $\notin$ CT(n) $\wedge$ pid $\notin$ pdtCache(CT(n)). In this case, we can use I.H. and infer that $\nexists L \in$ Comp(H(n, strLists)), Qualified(pid, Q, KW, L) = true. This leads to the conclusion $\nexists L \in$ Comp(H(n+1, strLists)), Qualified(pid, Q, KW, L) = true because Comp(H(n+1, strLists)) $\subseteq$ Comp(H (n+1, strLists)).

Case 1.2: pid $\in$ CT(n) $\vee$ pid $\in$ pdtCache(CT(n)). In this case, since pid $\notin$ CT(n+1) $\wedge$ pid $\notin$ pdtCache(CT(n+1)), we need to discuss when pid is removed at loop # n+1.

First assume pid ∈ CT(n) and assume at loop # t, pid is never temporarily copied to any pdt cache. Intuitively, this case indicates that pid does not satisfy the descendant restrictions.

By the algorithm there exists a node pn in the left most path of CT(n+) and pn.id = pid. By the algorithm pn must be removed by line 34 in Fig. 15. By $pid \notin$ CT(n+1), we can infer that $\forall q \in$ pn.CTQNodes, $\exists ch \in$ MC(q.CTQNode), q.DM[ch] = 0. Further, we remove pn only when pn.HasChild = false. Also, by line 15 in Fig. 15, we have already added next minimum ids corresponding to the left most path. Also, for all paths in the lists, the next minimum IDs are greater than their respective IDs in the current CT because they are ordered ID lists. This implies that $\forall L \in$ Comp (H(n+1, strLists)), if l ∈ L and l.QNode = ch, and if lid is the next id in l, then we know that Prefix(l.QNode, lid, pn.QNode) is greater than pn.id, and hence q. DM[ch] will never be set to be 1. Therefore by Lemma 3, we know that $\forall L \in$ Comp(H(n+1, strLists)), pid ∉ CE(pn.CTQNode, T(L).root)), and hence Qualified(pid, Q, KW, L) = false.

Second, if pid ∈ pdtCache(CT(n)) or pid is in CT(n) but was later copied to pdt cache of some nodes when we are at loop # n+1. Assume pn is the node in the pdt cache s.t. pn.id = pid, and assume pn ∈ cn.pdtCache. For simplification, we only consider the case where pn is removed when we process pn. Intuitively, this case indicates that pid does not satisfy the ancestor restrictions.

This is handled by line 26 in Fig. 15. Therefore we know that before we remove pn, pn.PL ={cn} and $\exists ch \in$ MC(cn), cn.DM[ch] = 0. By Lemma 3 and using the same argument as in the first case, we know that $\forall L \in$ Comp(H(n+1, strLists)), Qualified(cn.id, Q, KW, L) = false. Hence $\forall L \in$ Comp(H(n+1, strLists)), pn does not satisfy the ancestor restrictions, and therefore $\forall L \in$ Comp(H(n+1, strLists)), Qualified(pid, Q, KW, L) = false.

Case 2: pid ∈ Prefix(H(n+1,strLists)) - Prefix(H(n, strLists)). Note that in the algorithm, we first add ids in H(n+1, strLists) - H(n, strLists) (line 15 in Fig. 15) and then process the left most path, therefore if CT(n') is the intermediate candidate tree after we add new ids to CT(n), then pid ∈ CT(n') and we can use the same argument in Case 1.2 to show that the lemma holds. The full proof is skipped here.

Appendix C: Proofs of correctness of PrepareList

By Lemma 1, we know that once we exit the loop and the candidate tree becomes empty, all qualified ids w.r.t to strLists are captured in PDT and PDT only contains qualified ids w.r.t to strLists. In other words, if GenPDT is the PDT that is produced upon termination of the loop, then GenPDT = PDT(Q, KW, {Q.root ⇒ T(strLists).root}). We just need the following final lemma to show Theorem 2 is true.

We first show two supporting lemmas.

Given a QPT Q, a node n∈Q, we say RootToLeaf(n, Q) is the path starting from the root node of Q and ends on n, then we can show the following lemma:

Lemma 11 (PathIndex) *Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$, then $\forall q \in QPT$, $\forall n \in D$, $n \in PE(q$, {Q.root ⇒ d}) ⇒ id(n) ∈ d.PathIndex.LookUp(RootToLeaf(q, Q)).*

Proof We prove the lemma by inductions on the depth of q.

Base case: depth = 0. In this case, q is the root node of Q and RootToLeaf(q, Q) = {q}. Hence d.PathIndex.LookUp(RootToLeaf(q, Q)) = {id(n)| tag(n) = q.tag ∧ $\forall p \in$ q. Predicates, satisfies(n, q)}. Therefore d. PathIndex.LookUp(RootToLeaf
(q, Q)) is a superset of PE(q, {Q.root ⇒ d}. Hence the lemma holds.

Induction hypothesis: Assume the lemma holds for q of depth ≤ d, we need to show the lemma for q of depth d + 1.

Assume pq is the parent of q. We now show the case where (pq, q, '/', ann) ∈ Q, and the case where (pq, q, '//', ann) ∈ Q can be shown similarly.

By definition, $\forall n \in$ PE(q, {Q.root ⇒ d}), $\exists np \in$ PE(q, {Q.root ⇒ d}), parent(np, n). By I.H., we can infer that $\forall n \in$ PE(q, {Q.root ⇒ d}), $\exists pid \in$ d.PathIndex.LookUp(RootToLeaf(np, Q)), parent(pid, id(n)). Therefore we can infer that $\forall n \in$ PE(q, {Q.root ⇒ d}), $id(n) \in$ d.PathIndex. LookUp (RootToLeaf(q, Q)).

Hence the lemma holds.

Lemma 12 (CandidateElements) *Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), $\forall q \in Q, \forall n \in D$, n ∈ PE(q, {Q.root ⇒ d}) ⇒ n ∈ CE(q, {Q.root ⇒ T(strLists).root}).*

Proof We prove the lemma by induction on the depth of q.

Base case: q is the leaf node. In this case, $\forall n$ ∈PE(q, {Q.root ⇒ d}). Since q does not have children nodes, we will issue d.PathIndexLookUp(RootToLeaf(q, Q)). Therefore by Lemma 11, we can infer that id(n) ∈ strLists. Hence by the definition, n ∈ CE(q, {Q.root ⇒ T(strLists).root}).

Induction hypothesis: Assume the lemma holds for q of depth ≥ d, we need to show the lemma for q of depth d−1.

If MC(q) = ∅, then by the algorithm we will issue d.PathIndex. LookUp(RootToLeaf(q)). Hence similar to the base case we can show the lemma holds. Otherwise by definition of PDT and $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $\forall cq \in$ MC(q), (q, cq, '/', m) $\in$ Q $\Rightarrow \exists nc \in$ PE(cq, {Q.root $\Rightarrow$ d}), parent(n, nc) $\wedge$ (q, cq, '//', m) $\in$ Q $\Rightarrow \exists nc \in$ PE(cq, {Q.root $\Rightarrow$ d}), anc(n, nc).

Hence by I.H., we know that $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $\forall cq \in$ MC(q), (q, cq, '/', m) $\in$ Q $\Rightarrow \exists nc \in$ CE(cq, {Q.root $\Rightarrow$ T(strLists).root}), parent(n, nc) $\wedge$ (q, cq, '//', m) $\in$ Q $\Rightarrow$ $\exists nc \in$ CE(cq, {Q.root $\Rightarrow$ T(strLists).root}), anc(n, nc).

Further, by the definition of Dewey ID, we know that $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), id(n) $\in$ T(d), $\forall cq \in$ MC(q), (q, cq, '/', m) $\in$ Q $\Rightarrow \exists nc \in$ CE(cq, {Q.root $\Rightarrow$ T(strLists).root}), parent(n, nc) $\wedge$ (q, cq, '//', m) $\in$ Q $\Rightarrow \exists nc \in$ CE(cq, {Q.root $\Rightarrow$ T(strLists).root}), anc(n, nc).

Therefore $n \in$ CE(q, {Q.root $\Rightarrow$ T(strLists).root}).

Hence the lemma holds.

Lemma 13 (PrepareList) *Given a set of keywords KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root)$=d and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), $\forall q \in Q$, $\forall n \in D$, $n \in PE(q, \{Q.root \Rightarrow d\}) \Leftrightarrow n \in PE(q, \{Q.root \Rightarrow T(strLists).root\})$.*

Proof "$\Rightarrow$"
We prove this direction by induction on the depth of q.

Base case: depth = 0. In this case, q is the root node of the QPT. By definition, PE(q, {Q.root $\Rightarrow$ d}) = CE(q, {Q.root $\Rightarrow$ d}), and PE(q, {Q.root $\Rightarrow$ T(strLists).root}) = CE(q, {Q.root $\Rightarrow$ T(strLists).root}). By Lemma 12, we know PE(q, {Q.root $\Rightarrow$ d}) $\subseteq$ CE(q, {Q.root $\Rightarrow$ T(strLists).root}), and hence PE(q, {Q.root $\Rightarrow$ d}) $\subseteq$ PE(q, {Q.root $\Rightarrow$ T(strLists).root}). Therefore the lemma holds in the base case.

Induction Hypothesis: Assume the lemma holds for q of depth $\leq$ d, now we show the lemma also holds for q of depth d+1.

Assume p is the parent node of q in Q. We show the case where (p, q, '/', ann) $\in$ Q, the case where (p, q, '//', ann) $\in$ Q can be shown similarly.

First, by Lemma 12, $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $n \in$ CE(q, {Q.root $\Rightarrow$ T(strLists).root}). Then by definition, we know that $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $\exists np \in$ PE(p, PE(q, {Q.root $\Rightarrow$ d}), parent(np, n). Hence by I.H. on nq, we know that $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $\exists np \in$ PE(p, {Q.root $\Rightarrow$ T(strLists).root}) parent(np, n). Hence $\forall n \in$ PE(q, {Q.root $\Rightarrow$ d}), $n \in$ CE(q, {Q.root $\Rightarrow$ T(strLists).root}). $\wedge$ $\exists np \in$ PE(p, {Q.root $\Rightarrow$ T(strLists).root}) parent(np, n).

Therefore $n \in$ PE(q, {Q.root $\Rightarrow$ T(strLists).root}).

Hence the lemma holds.

"$\Leftarrow$"
This direction follows from Lemma 2 because $\forall id \in$ strLists, $id \in$ D. Hence the full proof is skipped.

References

1. Aboulnaga, A., Naughton, J.F., Zhang C.: Generating synthetic complex-structured XML data. In: WebDB, pp. 79–84 (2001)
2. Al-Khalifa, S., Yu, C., Jagadish, H.V.: Querying structured text in an XML database. In: SIGMOD (2003)
3. Amer-Yahia, S. et al.: Structure and content scoring for XML. In: VLDB (2005)
4. Theobald, A., Weikum, G.: The index-based XXL search engine for querying XML data with relevance rankings (2002)
5. Baeza-Yates, R., Ribeiro-Neto, B.: Modern information retrieval. ACM Press, New York (1999)
6. Bhaskar, A., et al.: Quark: an efficient XQuery full-text implementation. In: SIGMOD (2006)
7. Botev, C., Shanmugasundaram, J.: Context-sensitive keyword search and ranking for XML. In: WebDB (2005)
8. Bressan, S., Catania, B., Lacroix, Z., Li, Y.G., Maddalena, A.: Accelerating queries by pruning XML documents. Data Knowl. Eng. **54**(2), 211–240 (2005)
9. Carey, J.M.: XPERANTO: middleware for publishing object-relational data as XML documents. In: VLDB, pp. 646–648 (2000)
10. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. VLDB J. **11**(4), 354–379 (2002)
11. Chaudhuri, S., Gravano, L., Marian, A.: Optimizing top-k selection queries over multimedia repositories. IEEE Trans. Knowl. Data Eng. **16**(8), 992–1009 (2004)
12. Chen, Z., et al.: Index structures for matching XML twigs using relational query processors. Data Knowl. Eng. **60**(2), 283–302 (2007)
13. Chen, Z., Jagadish, H.V., Lakshmanan, L.V.S., Paparizos, S.: From tree patterns to generalized tree patterns: on efficient evaluation of XQuery. In: VLDB (2003)
14. Cho, S.: Indexing for XML siblings. In: WebDB (2005)
15. Christophides, V., Cluet, S., Simeon, J.: On wrapping query languages and efficient XML integration. In: SIGMOD (2000)
16. Curtmola, E., Amer-Yahia, S., Brown, P., Fernandez, M.: GalaTex: a conformant implementation of the XQuery full-text language. In: XIME-P (2005)
17. Diao, Y., Fischer, P., Franklin, M., To, R.: YFilter: efficient and scalable filtering of XML documents. In: ICDE (2002)
18. Fagin, R.: Combining fuzzy information from multiple systems. In: PODS (1996)
19. Fahl, G., Risch, T.: Query processing over object views of relational data. VLDB J **6**(4), 261–281 (1997)
20. Fernandez, M.F., Tan, W.C., Suciu, D.: SilkRoute: trading between relations and XML. Comput. Netw. **33**(1-6), 723–745 (2000)
21. Fuhr, N., Großjohann, K.: XIRQL: a query language for information retrieval in XML documents (2001)
22. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD (2003)
23. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB (2003)
24. Hristidis, V., Papakonstantinou, Y.: Discover: keyword search in relational databases. In: VLDB (2002)
25. Ilyas, I.F., et al.: Rank-aware query optimization. In: SIGMOD (2004)
26. Jagadish, H.V., et al.: TIMBER: a native XML database. VLDB J. **11**(4), 274–291 (2002)
27. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: ICDE (2004)
28. Marian, A., Siméon, J.: Projecting XML documents. In: VLDB (2003)
29. Mass, Y., et al.: JuruXML—an XML retrieval system at INEX'02. In: INEX (2002)

30. Myaeng, S.-H., Jang, D.-H., Kim, M.-S., Zhoo, Z.-C.: A flexible model for retrieval of SGML documents. In: SIGIR (1998)
31. Naughton, J.F., et al.: The niagara internet query system. IEEE Data Eng. Bull. **24**(2), 27–33 (2001)
32. O'Neil, P., et al.: ORDPATHs: insert-friendly XML node labels. In: SIGMOD (2004)
33. Paparizos, S., Wu, Y., Lakshmanan, L.V.S., Jagadish, H.V.: Tree logical classes for efficient evaluation of XQuery. In: SIGMOD. ACM Press, New York, pp. 71–82 (2004)
34. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS (2001)
35. Salton, G.: Automatic text processing: the transaction, analysis and retrieval of information by computer. Addison-Wesley, Reading (1989)
36. Schott, S., Noga, M.L.: Lazy XSL transformations. In: DocEng 2003. ACM Press, Grenoble (2003)
37. Shanmugasundaram, J., et al.: Querying XML views of relational data. In: VLDB (2001)
38. Shao, F., et al.: Efficient ranked keyword search over virtual XML views, technical report TR2007-2077, Cornell University (2007)
39. Shao, F., Guo, L., Botev, C., Bhaskar, A., Chettiar, M.M.M., Yang, F., Shanmugasundaram, J.: Efficient keyword search over virtual XML views. In: VLDB, pp. 1057–1068 (2007)
40. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: compressing and Indexing Documents and Images. Morgan Kaufmann Publishers, San Francisco (1999)
41. Yoshikawa, M., Amagasa, T.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Inter. Tech. **1**(1), 110–141 (2001)
42. Zhang, C., et al.: On supporting containment queries in relational database management systems. In: SIGMOD (2001)
43. Zobel, J., Moffat, A.: Exploring the similarity space. SIGIR Forum **32**(1), 18–34 (2001)