Efficient Keyword Search over Virtual XML Views

Feng Shao and Lin Guo and Chavdar Botev
and Anand Bhaskar and Muthiah Chettiar and Fan Yang
Cornell University, Ithaca, NY 14853
{fshao, guolin, cbotev, yangf}@cs.cornell.edu; {ab394, mm376}@cornell.edu

Jayavel Shanmugasundaram
Yahoo! Research, Santa Clara, CA 95054
jaishan@yahoo-inc.com

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 [5] open-source XML database system indicates that the proposed approach is scalable and efficient.

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 [4, 32] 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 [32] 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., [8, 15]), 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.

VLDB '07, September 23-28, 2007, Vienna, Austria.

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 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 number of the book, and then nesting the reviews under the book (Figure 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., [7, 13, 18]) 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 vir-

[1] http://my.yahoo.com
[2] http://www.microsoft.com/sharepoint

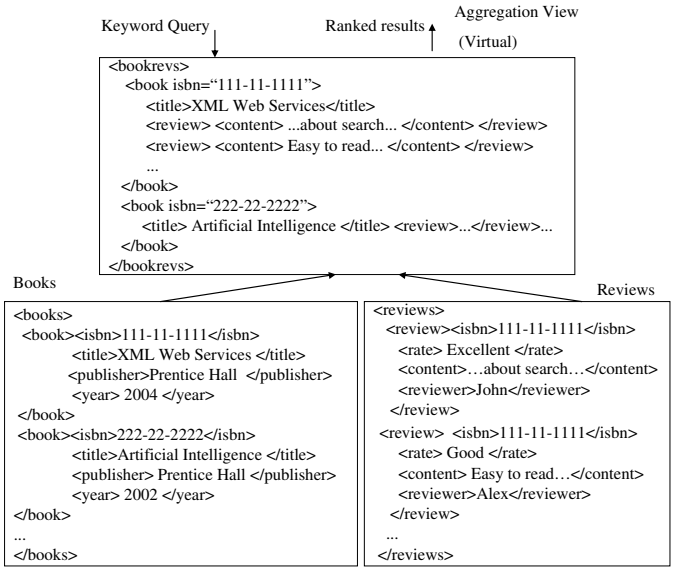

Figure 1: An XML view associating books & reviews

tual 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.

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 Figure 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 they *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 number, 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 number) 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 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 GTP [11] with TermJoin [1], 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 10 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 [26]; 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 (Section 3).
- Efficient algorithms for generating pruned XML elements needed for query evaluation and scoring, by solely using indices (Section 4).
- Evaluation and comparison of the proposed approach using the 500MB INEX dataset[3] (Section 5).

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 [32]. Generalizing the proposed approach to deal with more sophisticated XML scoring functions (e.g., [2, 20, 27]) is another interesting direction for future work.

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

```
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
```

Figure 2: Keyword Search over XML view

2. BACKGROUND & 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. Each element can have attributes and values, in addition to nested subelements. Figure 1 shows an example XML document representing books with nested reviews. Each ⟨*book*⟩ element has ⟨*title*⟩ and ⟨*review*⟩ subelements nested under it. The ⟨*book*⟩ element also has the isbn attribute whose value is "111-11-1111". For ease of exposition, we treat attributes as though they are subelements. 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 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 Figure 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 [2, 3, 19, 20, 27]. As mentioned in Section 1, in the paper we focus on the commonly used TF-IDF method proposed in the context of XML documents [19]. 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 [39].

2.3 Problem Definition

We use a set of keywords $Q = \{k_1, k_2, ..., k_n\}$ to represent a keyword search query, and define the problem of keyword search over views as follows.

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

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

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-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 set of k elements with highest scores in *RES(Q,V,D)*, where we break ties arbitrarily.

The above definition captures the result of *conjunctive* ranked keyword search queries over views. Our system also supports disjunctive queries which can be defined similarly.

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., [5, 14, 24, 29]). 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., [14, 29]), materialize the entire view before evaluating the keyword search query (e.g. [5]), or do not support such queries efficiently (e.g., [24]), as verified in our performance study (Section 5).

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 Figure 3). The modified query execution path (depicted by dashed lines in Figure 3) is as follows. On detecting a keyword search query over a view that satisfies certain conditions (clarified

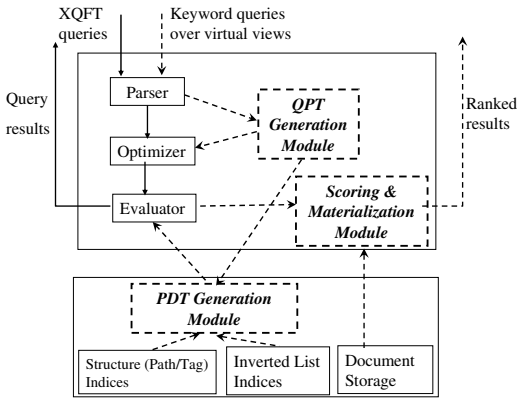

Figure 3: Keyword query processing architecture

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 GTP [11], 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, 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., [12]). We refer the reader to [35] for the supported grammar.

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.

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 [20] and update [30] 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 Figure 4(a).

Another important aspect is XML indexing. At a high-

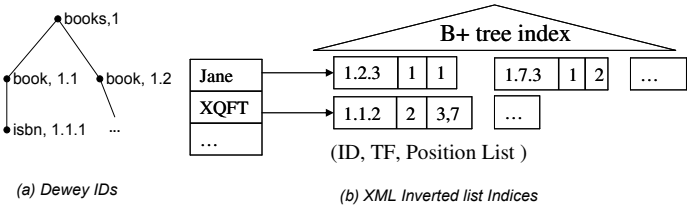

Figure 4: Illustrating XML Storage & Indices

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

Figure 5: XML path indices

level, there are two types of XML indices: path indices and inverted list indices (these indices can sometimes be combined [25]). 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 [10, 37] for efficient probes. Figure 5 shows the path index for the document in Figure 1. 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 *IDList*, 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., [20, 28, 38]) 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 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 (Figure 3) generates QPTs

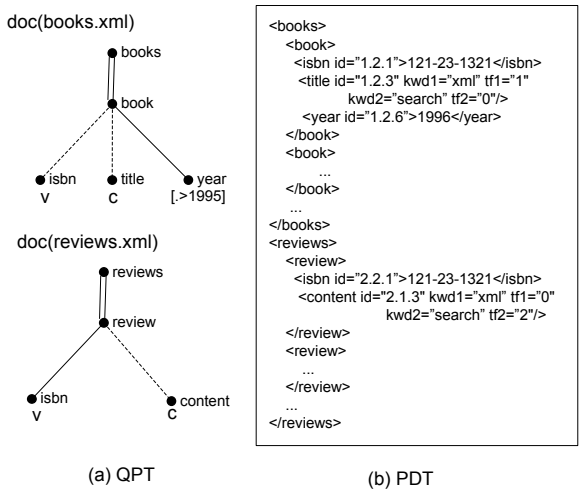

Figure 6: QPTs and PDTs of *book* and *review*

from an XML view. We illustrate the QPT using the view shown in Figure 2. In order to *evaluate* this view query, we only need a small subset of the data, such as the isbn numbers of books and isbn numbers 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) [11], 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. 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 6(a) 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 Figure 6(a) 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 Figure 6(a), 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 number; the edge between *review* and *isbn* is mandatory, because a review is of no relevance to query execution unless it has an isbn number (otherwise, it does not join 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 node is required to evaluate the view. In our example, the 'isbn' node in both the book and review QPT is 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 (Figure 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, Figure 6(b) shows the PDT of the *book* document for its QPT shown in Figure 6(a). 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 [26]. 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 Figure 6(a). 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 Section 5 show that our PDT generation is more than an order of magnitude faster then PROJ.

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

4.1 PDT Illustration & 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 three types of constraints: (1) an ancestor constraint, which requires that an ancestor element of e that 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) =*
  {$v \in D$ | tag name of v is n.tag $\wedge$
  the value of v satisfies all predicates in n.preds }.
- n is a non-leaf node in Q: *CE(n, D) =*
  {$v \in D$ | tag name of v is n.tag $\wedge$ for every edge e in Q, if e.parent is n and e.ann is 'm' (mandatory), then $\exists ec \in$ CE(e.child, D) such that
  (a) e.axis = '/' $\Rightarrow$ v is the parent of ec, and
  (b) e.axis = '//' $\Rightarrow$ v is an ancestor of ec }

Definition 1 recursively captures the descendant constraints from bottom up. For example, in Figure 6(a), 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) =*
  {$v \in D$ | v is in CE(n, D) $\wedge$
  for every edge e in Q, if e.child is n,
  then $\exists vp \in$PE(e.parent, D) such that
  (a) e.axis = '/' $\Rightarrow$ vp is the parent of v, and
  (b) e.axis = '//' $\Rightarrow$ 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 Figure 6(a), 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, 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.*

```
 1: PrepareLists (QPT qpt, PathIndex pindex, InvertedIndex
    iindex, KeywordSet kwds): (PathLists, InvLists)
 2:   pathLists ← ∅; invLists ← ∅
 3:   for Node n in qpt do
 4:     p ← PathFromRoot(n); newList ← ∅
 5:     if n has no mandatory child edges then
 6:       n.visited ← true
 7:       if n has a 'v' annotation then
 8:         {Combining retrieval of IDs and values}
 9:         newList ← (n, pindex.LookUpIDValue(p))
10:       else
11:         newList ← (n, pindex.LookUpID(p))
12:       end if
13:     end if
14:     {Handle 'v' nodes with mandatory child edges}
15:     if n.visited = false ∧ n has a 'v' annotation then
16:       newList ← (n, pindex.LookUpIDValue(p))
17:     end if
18:     if newList ≠ null then pathLists.add(newList)
19:   end for
20:   for all k in kwds do
21:     invLists ← invLists ∪ (k, sindex.lookup(k))
22:   end for
23:   return (pathLists, invLists)
```

Figure 7: Retrieving IDs and values

- *$N = \cup_{q \in Q}$ PE(q, D), and nodes in N are associated with required values, tf values and byte lengths.*
- *E = {(p, c) | p, c are in $N \wedge$ p is an ancestor of c $\wedge$ $\nexists q \in N$ 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 a fixed number of index lookups 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 fixed number of probes on path indices. First, we issue index lookups for QPT nodes that do not have mandatory child 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 [11]. 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 Figure 6(a), we only need to perform three index lookups on path indices (shown in Figure 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

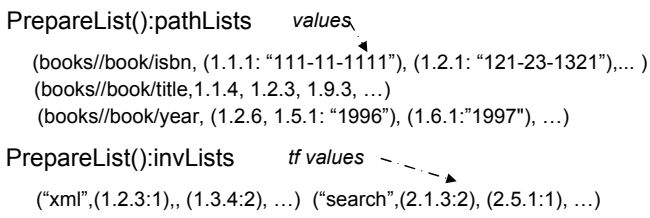

Figure 8: Results of PrepareLists()

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 Figure 6(a) and the path indices in Figure 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 Figure 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, our algorithm also returns the relevant inverted index indices to obtain scoring information.

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, and Inverted Index as input, and first issues a lookup on path indices for each QPT node that has no mandatory child edges (lines 5- 13). It then identifies nodes that have a 'v' annotation (lines 9 & 16), and for each path from the root to one of these nodes, the algorithm issues a query to obtain the values and IDs (by only specifying the path). Finally, the algorithm looks up inverted lists indices and retrieves the list of Dewey IDs containing the keywords along with tf values (lines 20-22). Figure 8 shows the output of PrepareList for the book QPT (Figure 6(a)). 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 Figure 6(b) by merging the lists shown in Figure 8.

The main challenges in designing such an algorithm are: (1) we must enforce complex ancestor and descendant constraints (described in Section 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 are at least one ID corresponding to each relevant QPT node for checking descendant constraints.

```
1:  GeneratePDT (QPT qpt, PathIndex pindex, KeywordSet
    kwds, InvertedIndex iindex): PDT
2:    pdt ← ∅
3:    (pathLists, invLists) ← PrepareLists(qpt, pindex, iindex,
      kwds)
4:    for idlist ∈ pathLists do
5:      AddCTNode(CT.root, GetMinEntry(idlist), 0)
6:    end for
7:    while CT.hasMoreNodes() do
8:      for all n ∈ CT.MinIDPath do
9:        q ← n.QPTNode
10:       if pathLists(q).hasNextID() ∧ there do not exist
          ≥ 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
```

Figure 9: Algorithm for generating PDTs

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 satisfies 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.

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

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 QN-

```
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
```

Figure 10: Algorithm for adding new CT nodes

```
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
```

Figure 11: Processing CT.MinIDPath

ode'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 Figure 12(a), which is created using IDs 1.1.1, 1.1.4, and 1.2.6, corresponding to paths in pathLists shown in Figure 8. First, 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

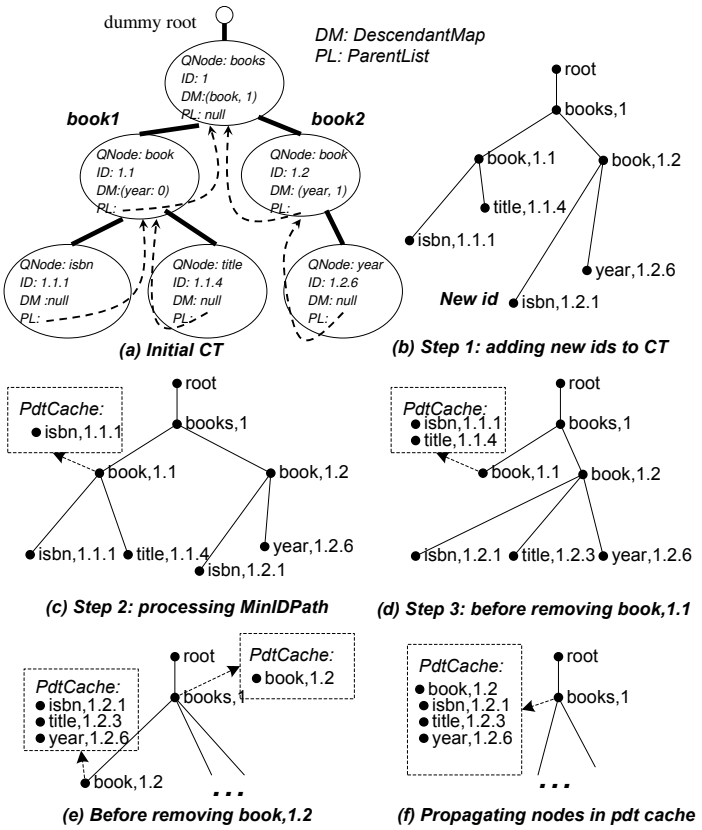

Figure 12: Generating PDTs

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.

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 12(a), which we just described, shows the initial CT for our example, which is created by adding minimum IDs of paths in pathLists shown in Figure 8. Note that for ease of exposition, our algorithm assumes each Dewey ID corresponds to 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 Section 4.2.2.1.

Description of the main loop

Next the algorithm enters the loop(lines 7-15 in Figure 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 Figure 12(a), this path is "books//book/isbn" and Figure 12(b) 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 (Figure 11, lines 2-4). We first check if the node satisfies the descendant constraints using values in its DM. In Figure 12(b), DM of the element "books" has value 1 in all entries, hence 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 12(c) 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 (Figure 11, line 7-20). For instance, in Figure 12(c), 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 12(d) 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 thus be propagated to the pdt cache of the ancestor. Figure 6(e) 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 that the invariant of the main loop holds.

4.2.2.1 Extensions and optimizations.

As mentioned earlier, when the QPT has repeating tag names, a single Dewey ID can match multiple QPT nodes. 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. In general, different QPT nodes capture different ancestor/descendant constraints. Hence they must be treated separately.

Further, there are two possible optimizations in the current algorithm. First, the 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). Second, to optimize the memory usage, we can output PDT nodes in document order (to external storage). We refer the reader to [35] for complete details and corresponding revisions to our algorithm.

4.2.2.2 Scoring & generating the results.

As shown in Figure 3, once the PDTs are generated (e.g., the PDT of our running example is shown in Figure 6(b)), 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 Figure 6(b)), 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 Section 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 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 content required for producing the results is retrieved.

4.3 Complexity and Correctness of Algorithms

The runtime of GeneratePDT is $O(Nqdf + Nqd^2 + Nd^3 + Ndkc)$ where N is the number of the IDs in pathLists, d is the depth of the document, q and f are the depth and 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 node; every QPT node can have d parent CT nodes and f mandatory child nodes. Nd^3 deter-

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

Table 1: Experimental parameters.

mines 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 of 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], and SIGMOD Record[5], and INEX), as also seen in our experiments.

We can prove the following correctness theorem (proofs are presented in [35]). If 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 Section 2.1.

THEOREM 4.1 (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)*
- *$\forall e \in Q(PDTDB)$, $e' \in Q(D)$, $I(e) = e' \Rightarrow$ PDTByteLength(e) = len(e') (The byte lengths of each element are identical)*
- *$\forall e \in Q(PDTDB)$, $e' \in Q(D)$, $I(e) = e' \Rightarrow (\forall k \in KW$, PDTTF(e,k) = tf(e',k)) (The term frequencies of each keyword in each element is identical)*

5. EXPERIMENTS

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

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

[4]http://dblp.uni-trier.de/xml/
[5]http://acm.org/sigmod/record/xml/

We evaluated the performance of four alternative approaches:
Baseline: materializing the view at the query time, and evaluating keyword search queries over views implemented using Quark.
GTP: GTP with TermJoin for keyword searches and implemented using Timber [1].
Efficient: our proposed keyword query processing architecture (Section 3.1) developed using Quark, with all optimizations and extensions implemented(Section 4.2.2.1).
Proj: techniques of projecting XML documents [26].

We have implemented scoring in **Efficient**. Recall that our score computation (Section 4.2.2.2) 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 experiments were run on an Intel 3.4Ghz P4 processors running Windows XP with 2GB of main memory. The reported results are the average of five runs.

5.2 Performance Results

5.2.1 Varying size of data

Figure 13 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 linearly 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 takes 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, Figure 13 shows that EFFICIENT performs ∼10 times faster than GTP. Note that Figure 13 only shows the time spent by GTP 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 GTP, including the time to perform the value join, was more than 5 minutes on the 100MB data set). GTP 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 post-processing are not included), its runtime is ∼15 times slower than EFFICIENT. The main reason is that PROJ scans

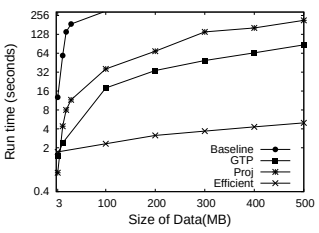

Figure 13: Varying size of data

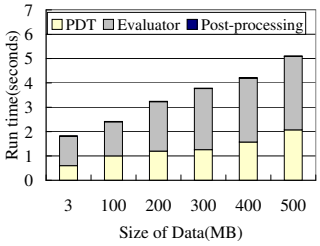

Figure 14: Cost of Modules

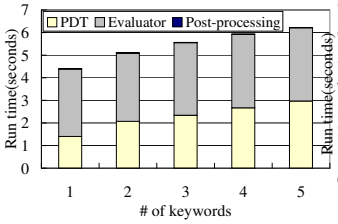

Figure 15: Varying # keywords

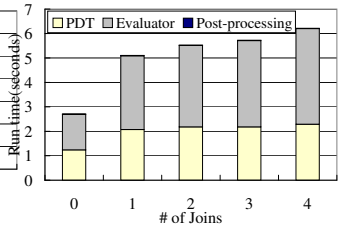

Figure 16: Varying the number of joins

base documents which leads to relatively poor scalability.

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

5.2.2 Evaluating Overhead of Individual Modules

Figure 14 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.

5.2.3 Varying other parameters

Varying # of keywords: Figure 15 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 16 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.

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 [35]) 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.

6. RELATED WORK

There has been a large body of work in the information retrieval community on scoring and indexing [21, 22, 32, 36]. 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 [6], 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., [7, 17, 34]), 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., [9, 31, 23]), 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 [11] with TermJoin [1] 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 (Section 5).

Finally, our PDT generation technique is related to the technique for projecting XML documents [26]. 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 Section 4 for other technical differences between the two approaches. Our technique is also related to the projection operator in Timber [24] and lazy XSLT transformation of XML documents [33], which, like PROJ, also access the base data for projection.

7. 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., [9, 16, 23]) 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.

8. ACKNOWLEGEMENTS

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.

9. REFERENCES

[1] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In *SIGMOD*, 2003.

[2] S. Amer-Yahia et al. Structure and Content Scoring for XML. In *VLDB*, 2005.

[3] A.Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Rankings . 2002.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[5] A. Bhaskar et al. Quark: An Efficient XQuery Full-Text Implementation. In *SIGMOD*, 2006.

[6] C. Botev and J. Shanmugasundaram. Context-Sensitive Keyword Search and Ranking for XML. In *WebDB*, 2005.

[7] M. J. Carey et al. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, 2000.

[8] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *VLDB Journal*, 11(4), 2002.

[9] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing Top-k Selection Queries over Multimedia Repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8), 2004.

[10] Z. Chen et al. Index Structures for Matching XML Twigs Using Relational Query Processors. *Data Knowl. Eng.*, 60(2):283–302, 2007.

[11] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *VLDB*, 2003.

[12] S. Cho. Indexing for XML Siblings. In *WebDB*, 2005.

[13] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *SIGMOD*, 2000.

[14] E. Curtmola, S. Amer-Yahia, P. Brown, and M. Fernandez. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. In *XIME-P*, 2005.

[15] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *ICDE*, 2002.

[16] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *PODS*, 1996.

[17] G. Fahl and T. Risch. Query Processing Over Object Views of Relational Data. *VLDB Journal*, 6(4).

[18] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1-6), 2000.

[19] N. Fuhr and K. Groβjohann. XIRQL: A Language for Information Retrieval in XML Documents. 2001.

[20] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[21] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, 2003.

[22] V. Hristidis and Y. Papakonstantinou. Discover: Keyword Search in Relational Databases. In *VLDB*, 2002.

[23] I. F. Ilyas et al. Rank-aware query optimization. In *SIGMOD*, 2004.

[24] H. V. Jagadish et al. TIMBER: A Native XML Database. *VLDB J.*, 11(4), 2002.

[25] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *ICDE*, 2004.

[26] A. Marian and J. Siméon. Projecting XML Documents. In *VLDB*, 2003.

[27] Y. Mass et al. JuruXML – an XML retrieval system at INEX'02. In *INEX*, 2002.

[28] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo. A Flexible Model for Retrieval of SGML Documents. In *SIGIR*, 1998.

[29] J. F. Naughton et al. The Niagara Internet Query System. *IEEE Data Eng. Bull.*, 24(2), 2001.

[30] P. O'Neil et al. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

[31] R.Fagin, A.Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[32] G. Salton. *Automatic Text Processing: The Transaction, Analysis and Retrieval of Information by Computer*. Addison Wesley, 1989.

[33] S. Schott and M. L. Noga. "lazy xsl transformations". In *DocEng 2003*, Grenoble, France, Nov 2003. ACM Press.

[34] J. Shanmugasundaram et al. Querying XML Views of Relational Data. In *VLDB*, 2001.

[35] F. Shao et al. Efficient Ranked Keyword Search over Virtual XML Views, Technical Report TR2007-2077, Cornell University. 2007.

[36] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[37] M. Yoshikawa and T. Amagasa. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Inter. Tech.*, 1(1), 2001.

[38] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD*, 2001.

[39] J. Zobel and A. Moffat. Exploring the Similarlity Space. *SIGIR Forum*, 32(1), 2001.