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

Jayavel Shanmugasundaram
Yahoo! Research

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 for queries over the virtual view, and the resulting scores and rank order are the same *as if* the view was materialized. Our performance evaluation using the INEX data set in the Quark [8] 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 [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., [11, 18]), 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.

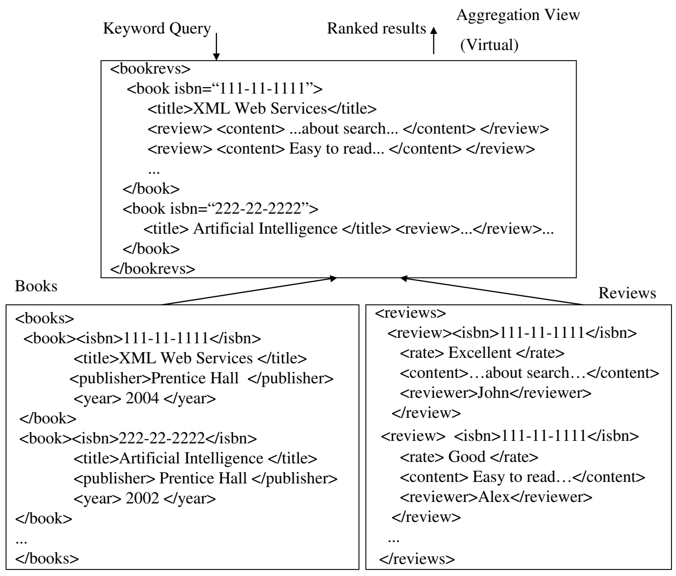
Personalized Views: Consider a large online web portal such as MyYahoo[1] that caters to millions of users. Since different users may have different interests, the portal may wish to provide a personalized view of the content to its users (such as books on topics of interest to the user along with their 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., [10, 16, 21]) 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

VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

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



```
<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>
```

Figure 1: An XML view associating books & reviews

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 tree*s (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 [14] 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 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 [30]; 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 over alternative approaches, an even more efficient strategy might be to avoid even 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, 24, 31]) 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 $\langle book \rangle$ element has $\langle title \rangle$ and $\langle review \rangle$ subelements nested under it. The $\langle book \rangle$ element also has the isbn attribute 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 interchangeable 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, 22, 24, 31]. As mentioned in Section 1, in the paper we focus on the commonly used TF-IDF method proposed in the context of XML documents [22]. 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: $score(e, Q) = \Sigma_{k \in Q}(tf(e, k) \times idf(k))$. The score can be further normalized using various methods proposed in the literature [40].

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

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., [8, 17, 28, 33]). 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., [17, 33]), materialize the entire view before evaluating the keyword search query (e.g. [8]), or do not support such queries efficiently (e.g., [28]), 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

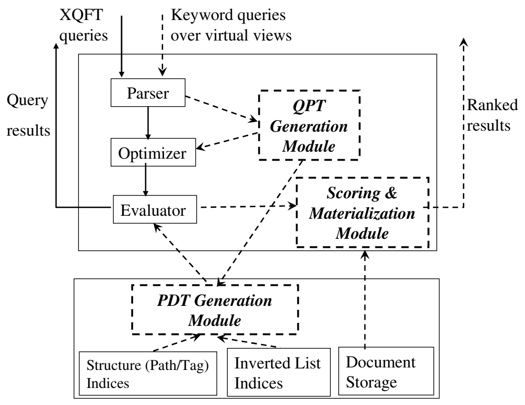


Figure 3: Keyword query processing architecture

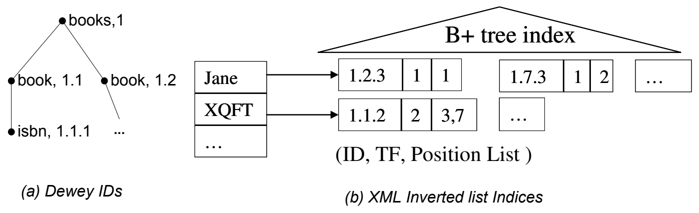


Figure 4: Illustrating XML Storage & Indices

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"	1.2.3, 1.7.3

Path-Values Table

Figure 5: XML path indices

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 [14], 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., [15]). We refer the reader to [37] 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 [24] and update [34] 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).

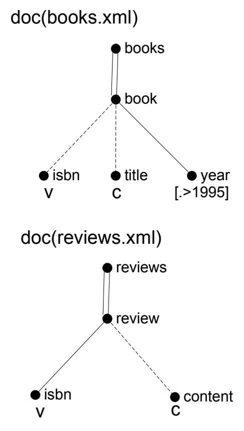
Another important aspect is XML indexing. At a high-level, there are two types of XML indices: path indices and inverted list indices (these indices can sometimes be combined [29]). Path indices are used to evaluate XML path and twig (i.e., branching path) queries. Inverted list indices are used to evaluate 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 [13, 38] 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 an element on the path. For each unique (Path, Value) pair, the table stores an *IDList* value, 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.

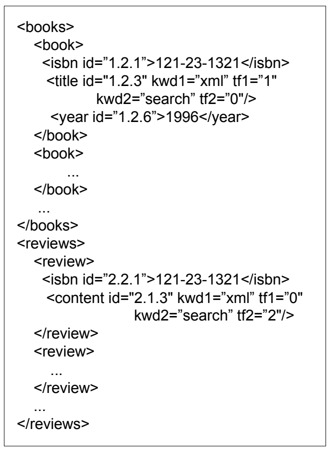
The second type of XML indices are inverted list indices. XML inverted list indices (e.g., [24, 32, 39]) typically store for each keyword in the document collection, the list of XML elements that *directly* contain the keyword. Figure 4 shows an example inverted list for our example document. In addition, 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



(a) QPT



```
<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>
```

(b) PDT

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) [14], 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 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 just 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 [30]. 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 has *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.*

```
PrepareLists (QPT $qpt$, PathIndex $pindex$, InvertedIndex $iindex$, KeywordSet $kwds$): (PathLists, InvLists)
  $pathLists \leftarrow \emptyset$; $invLists \leftarrow \emptyset$
  for Node $n$ in $qpt$ do
    $p \leftarrow PathFromRoot(n)$; $newList \leftarrow \emptyset$
    if n has no mandatory child edges then
      $n.visited \leftarrow true$
      if n has a 'v' annotation then
        {Combining retrieval of IDs and values}
        $newList \leftarrow (n, pindex.LookUpIDValue(p))$
      else
        $newList \leftarrow (n, pindex.LookUpID(p))$
      end if
    end if
    {Handle 'v' nodes with mandatory child edges}
    if $p.visited = false \wedge$ n has a 'v' annotation then
      $newList \leftarrow (n, pindex.LookUpIDValue(p))$
    end if
    if newList $\neq$ null then $pathLists.add(newList)$
  end for
  for all k in kwds do
    $invLists \leftarrow invLists \cup (k, sindex.lookup(k))$
  end for
  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, using the information gathered from indices, how do we efficiently generate the PDTs? 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 nodes in QPT 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 [14]. For instance, for the book QPT shown 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*, 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 conve-

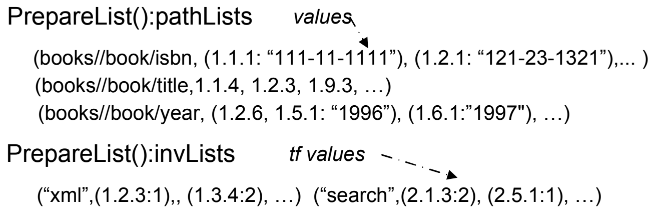

PrepareList():pathLists
values
(books//book/isbn, (1.1.1: “111-11-1111”), (1.2.1: “121-23-1321”),...)
(books//book/title,1.1.4, 1.2.3, 1.9.3, …)
(books//book/year, (1.2.6, 1.5.1: “1996”), (1.6.1:”1997"), …)
PrepareList():invLists
tf values
(“xml”,(1.2.3:1),, (1.3.4:2), …) (“search”,(2.1.3:2), (2.5.1:1), …)

Figure 8: Results of PrepareLists()

niently 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 return 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 of the lists produced by PrepareList and produces the PDT. The PDT satisfies the mutual 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

```
 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

shows the high-level pseudo-code of our algorithm which works as follows. The algorithm takes in an QPT, path index and inverted index of the document, and begins by invoking PrepareList 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.

Next the algorithm invokes the function CreatePDTNodes (line 14) and check 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 know there do not exist more of its descendant IDs in pathLists and 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 specifications.

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 which consists of candidate nodes for the result PDT. Every CT node *cn* 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.

```
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

- 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 is a candidate element with respect to c.
- PdtCache: the cache storing cn's descendants that satisfy descendant restrictions and 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 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

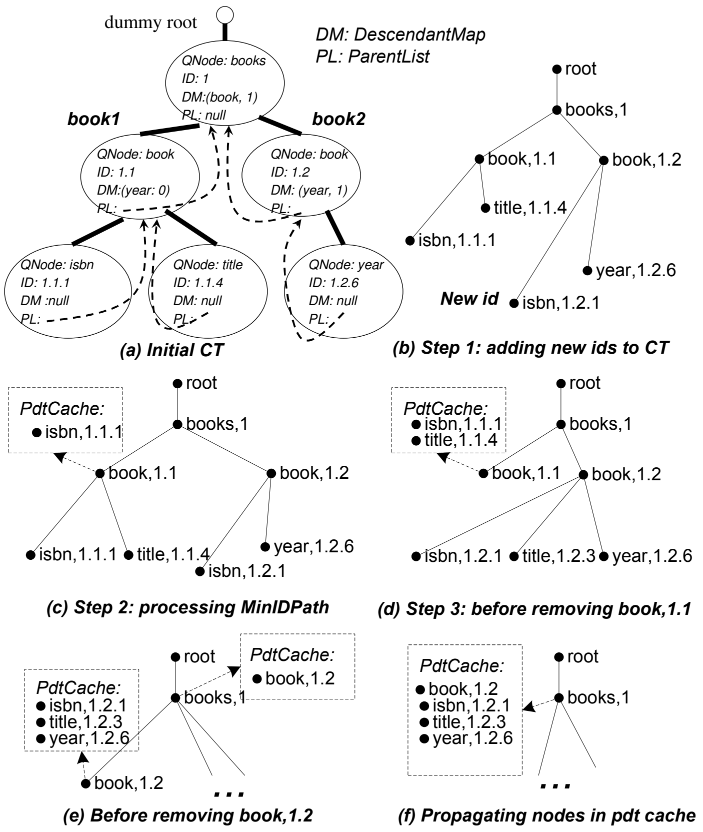


Figure 12: Generating PDTs

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 and bear in mind that they will be propagated along with the corresponding 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 it 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 corresponding inverted lists (line 10).

Figure 12(a), which we just described, is the initial CT for our running 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 more than one 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 specifications.

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). Last, we remove those nodes in MinIDPath that do not have any children from bottom up (Step 3). We now describe each step in more detail.

Step 1: adding new IDs In this step, the algorithm adds next IDs 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 its descendants that satisfy the descendant restrictions, and the checking of the ancestor restrictions is deferred until the node itself is being 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. Such node can be removed because as mentioned earlier, we can be certain that it will not satisfy the descendant restrictions (as in our example).

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 at all, and therefore can be removed (line 13). Otherwise the cache node still has other parents (which can be PDT nodes), and will 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.

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 the invariant of the main loop holds. 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.

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. This is because in general different QPT nodes capture different ancestor/descendant constraints, hence must be treated separately.

Further, there are two possibilities of optimizations in the current algorithm. First, the algorithm always creates and propagates 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 enforce the PDT nodes to be output in document order (to external storage). We refer the reader to Appendix E 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, 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 contents required for producing the results are 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 fanout 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

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.

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 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 Appendix F). 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 $D \in UD$, if PDTDB = {GeneratePDT(QPT, D.PathIndex, D.InvertedIndex, KW) | QPT $\in$ 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).

We evaluated the performance of four alternative approaches:
Baseline: materializing the view at the query time, and evaluating keyword search queries over view implemented using Quark.
GTP: GTP with TermJoin for keyword searches and implemented using Timber [2].
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 [30].

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, and hence we do not run separate experiments using the measure of precision-recall to evaluate the effectiveness of scoring.

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 a machine with a 3.4Ghz P4 CPU and 2GB memory running Windows XP. 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 only took 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 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, Figure 13 shows that Efficient performed ~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) in GTP. We do not report the overhead of the rest of query evaluation because it was 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 must access base data to obtain join values.

Finally, while Proj merely characterizes the cost of gener-

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

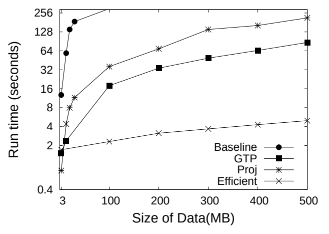


Figure 13: Varying size of data

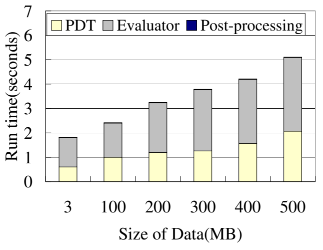


Figure 14: Cost of Modules

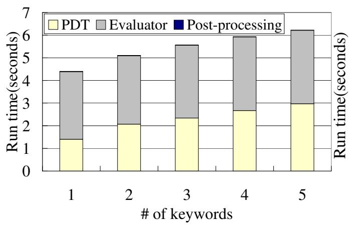


Figure 15: Varying # keywords

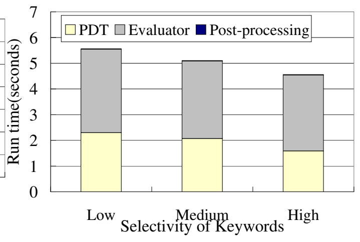


Figure 16: Varying selectivity of keywords

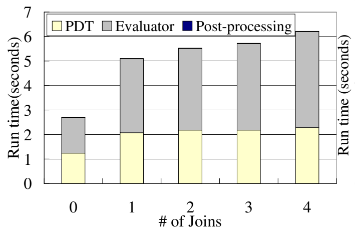


Figure 17: Varying the number of joins

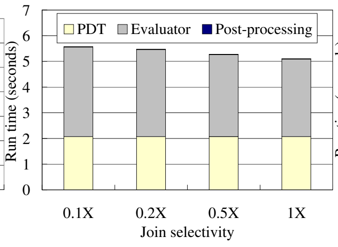


Figure 18: Varying the selectivity of joins

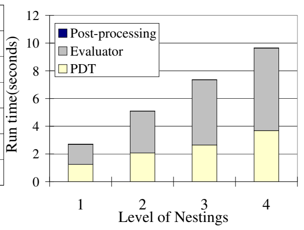


Figure 19: Varying the level of nestings

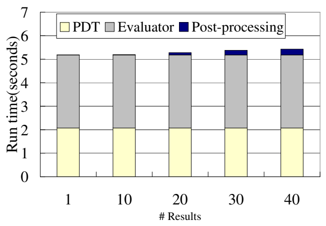


Figure 20: Varying the number of results

ating 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 full 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. Second, 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 slightly increases because the algorithm accessed a larger number of inverted lists to retrieve tf values, which introduces additional overhead when generating PDTs.

Varying selectivity of keywords: Figure 16 shows the performance results when varying the selectivity of the keywords. The run time increases slightly when the selectivity of keywords decreases. This is mainly because the overhead of generating PDTs increases – as the selectivity goes down, the length of the inverted list becomes larger which increases the I/O cost of retrieving tf values.

Varying # of joins: Figure 17 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 the selectivity of joins: Figure 18 shows the performance results when varying the selectivity of value joins in the view definition. As shown, the run time increases slightly when the selectivity decreases mainly because the cost of the query evaluation increases.

Varying the level of nestings: Figure 19 shows the performance results when varying the level of nestings in the view. This experiment shows that the run time increases linearly with the level of nestings, while the overhead of the query evaluator grows relatively faster than other modules.

Varying the number of results: Figure 20 shows the performance results when varying the number of results (i.e., K in top-K). As shown, the run time remains approximately the same because the overhead of storing and materializing additional results is nearly negligible.

Other results: We also vary the size of the view element and the performance results (available in [37]) show that our approach is efficient and scalable with increased size of elements. Second, the PDTs generated with respect to the data collection (500MB) are 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 [1, 6, 7, 24, 25, 26]. 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 [9, 23], 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., [10, 20, 36]), 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., [12, 19, 27]), 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 [14] 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 (Section 5).

Finally, our PDT generation technique is related to the technique for projecting XML documents [30]. 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.

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., [12, 19, 27]) 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 idf values, 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. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an xml database. In *SIGMOD*, 2003.

[3] S. Amer-Yahia et al. Structure and content scoring for xml. In *VLDB*, 2005.

[4] A.Theobald and G. Weikum. The index-based xxl search engine for querying xml data with relevance rankings. 2002.

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

[6] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[7] G. Bhalotia et al. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[8] A. Bhaskar et al. Quark: An efficient xquery full-text implementation. In *SIGMOD*, 2006.

[9] C. Botev and J. Shanmugasundaram. Context-sensitive keyword search and ranking for xml. In *WebDB*, 2005.

[10] M. J. Carey et al. XPERANTO: Middleware for publishing object-relational data as xml documents. In *The VLDB Journal*, 2000.

[11] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *VLDB Journal*, 11(4), 2002.

[12] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8), 2004.

[13] Z. Chen et al. Index structures for matching xml twigs using relational query processors. In *ICDEW '05*, 2005.

[14] 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.

[15] S. Cho. Indexing for xml siblings. In *WebDB*, 2005.

[16] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient xml integration. In *SIGMOD*, 2000.

[17] E. Curtmola, S. Amer-Yahia, P. Brown, and M. Fernandez. Galatex: A conformant implementation of the xquery full-text language, 2004.

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

[19] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[20] G. Fahl and T. Risch. Query processing over object views of relational data. *VLDB Journal: Very Large Data Bases*, 6(4).

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

[22] N. Fuhr and K. Grobjohann. Xirql: A language for information retrieval in xml documents. 2001.

[23] T. Grabs and H.-J. Schek. Powerdb-xml: A platform for data-centric and document-centric xml processing. In *Xsym*, 2003.

[24] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD*, 2003.

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

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

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

[28] H. V. Jagadish et al. Timber: A native xml database. *VLDB J.*, 11(4), 2002.

[29] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *ICDE*, 2004.

[30] A. Marian and J. Siméon. Projecting xml documents. In *VLDB*, 2003.

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

[32] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo. A flexible model for retrieval of sgml documents. In *SIGIR*, 1998.

[33] J. F. Naughton et al. The niagara internet query system. *IEEE Data Eng. Bull.*, 24(2), 2001.

[34] P. O'Neil et al. Ordpaths: Insert-friendly xml node labels. In *SIGMOD*, 2004.

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

[36] J. Shanmugasundaram et al. Querying XML views of relational data. In *VLDB*, 2001.

[37] F. Shao et al. Efficient ranked keyword search over virtual XML views (technical report), http://www.cs.cornell.edu/∼fshao/research.htm.

[38] 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.

[39] C. Zhang et al. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.

[40] J. Zobel and A. Moffat. Exploring the Similarlity Space. *SIGIR*

Forum, 32(1), 2001.

APPENDIX

A. THE SUPPORTED GRAMMAR

The supported grammar in our keyword query processing architecture 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(Name) | VAR | .
          | (fn:doc(Name) | 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 'in' 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)*
```

It is easy to verify that the view in our running example 2 conforms to the above grammar.

B. QPT GENERATION ALGORITHMS

In this section, we present and prove the correctness of the algorithm GenerateQPT.

The main challenge lies in correctly determining the shape of the tree and the associated annotations for arbitrarily complex views that conform to the grammar presented in Appendix A.

Our algorithm (Figure 21) works as follows. The recursive function `GenerateQPT` takes in the current expression (e) and returns a set of QPTs generated. The algorithm is initially invoked with e set to be the expression that defines the view. When processing an expression, the algorithm also sets the node annotion for the nodes generated in QPTSet indicating whether the corresponding expression contributes to the content of view. Note in the algorithm, the edge label 'm' indicates a mandatory edge and the edge label 'o' indicates an optional edge.

Now we use our running example to walk through the algorithm. For ease of exposition, we unfold the recursive call and illustrate the construction of QPTs from bottom up. Figure 23 shows the process of creating the QPT for nodes in books.xml at each phase. Initially, we call lines 6-9 in Figure 24 and generates the PDT for the expression "\$book/year > 1995". Figure 23(a) show the QPT at this point. Note that by line 31 in Figure 22, the predicate is now associated with the leaf QPT node. Next, we generate the QPT for the expressions in the return clause (line 12 in Figure 24). As shown in Figure 23(b), two additional twigs are created with optional edges. The intuition is that by the semantics of FLOWR expression, the existence of the parent element "\$book" does not depend on the existence of "isbn" or "title". This is in contrast to the edge create in step 1 in which case the existence of "\$book" is restricted by

```
1:  GenerateQPT (Expr $e$) : QPTSet
2:    if $e$ istype $PathExpr$ then
3:      if $e$ istype fn:doc(Name) or VAR or '.' then
4:        {A new QPT is created for the expression}
5:        $n \leftarrow (e,\{\})$
6:        $V\text{-}AnnMap[n] \leftarrow false$ , $C\text{-}AnnMap[n] \leftarrow true$
7:        return $\{(\{n\},\{\},n)\}$
8:      else if $e$ istype (fn:doc(Name) | VAR | '.') '/'
        PathTailExpr then
9:        {Q} ← GenerateQPT((fn:doc(Name) | VAR | '.'))
10:       V ← $Q.V$; E ← $Q.E$
11:       for all tempQpt in GenerateQPT(PathTailExpr)
          do
12:         for all (tempQpt.root, n, axis, ann) in
            tempQpt.E do
13:           E.add(child, n, axis, ann); V.add(n);
14:         end for
15:       end for
16:       return {(V, E, Q.root)}
17:     else if $e$ istype (fn:doc(Name) | VAR | '.') //
        PathTailExpr then
18:       {Similar to (fn:doc(Name) | VAR | '.') /
          PathTailExpr}
19:     else
20:       {e is PathExpr '[' PredExpr ']'}
21:       $qptSet \leftarrow \emptyset$
22:       $predQptSet$ ← GenerateQPT(PredExpr)
23:       for all $pathQpt$ in GenerateQPT(PathExpr) do
24:         for all $predQpt$ in $predQptSet$ where
            predQpt.root is '.' do
25:           pathQpt.E ← pathQpt.E ∪ $\{(l, n, axis, ann)$
              $|l \in Leaf(pathQpt)$
              $\wedge(predQpt.root, n, axis, ann) \in predQpt.E\}$
26:         end for
27:         $qptSet \leftarrow qptSet \cup \{pathQpt\}$
28:       end for
29:       predSet ← predSet - $\{Q \in predSet|Q.root\ is\ '.'\}$
30:       return qptSet ∪ predSet
31:     end if
32:   else if $e$ is PathTailExpr then
33:     {refer to Figure 22}
34:   else if $e$ istype PredExpr then
35:     {refer to Figure 22}
36:   else if $e$ istype 'if' Expr1 'then' Expr2 else 'Expr3'
      then
37:     for all $Q \in GenerateQPT(Expr1)$, $n \in Q.V$ do
38:       C-AnnMap[n] = false;
39:     end for
40:     return $GenerateQPT(Expr1)$ ∪
        $GenerateQPT(Expr2)$ ∪ $GenerateQPT(Expr3)$
41:   else if $e$ istype FLOWRExpr then
42:     Refer to figure 24.
43:   else if $e$ istype FunctionCall then
44:     {e=QName "(" (PathExpr ("," PathExpr)*)? ")" }
45:     $qptSet \leftarrow \emptyset$
46:     $funcDecl$ ← GetFunctionDecl(QName)
47:     $funcQptSet$ ← GenerateQPT(funcDecl.Expr)
48:     for all PathExpr in e do
49:       $pathQptSet$ ← GenerateQPT(PathExpr)
50:       index ← e.GetIndex(PathExpr)
51:       VAR ← funcDel.ParamList[index]
52:       for all pathQpt in pathQptSet do
53:         for all funcQpt in funcQptSet where
            funcQpt.root is VAR do
54:           pathQpt.E ← pathQpt.E ∪ $\{(l, n, axis, ann)$
              $|l \in Leaf(pathQpt)$
              $\wedge(funcQpt.root, n, axis, ann) \in funcQpt.E\}$
55:         end for
56:         qptSet ← qptSet ∪ {pathQpt}
57:       end for
58:       funcQptSet ← $funcQptSet$
          $-\{Q \in funcQptSet|Q.root\ is\ VAR\}$
59:     end for
60:     qptSet ← qptSet ∪ funcQptSet
61:   end if
```

Figure 21: Algorithm for producing Query Pattern Tree (QPT) from a keyword query

```
GenerateQPT (Expr $e$) : QPTSet
  if $e$ is $PathTailExpr$ then
    if $e$ is 'TAGNAME' then
      {Create a '.' node with a child node for 'TAGNAME'}
      $root$ ← ('.',{})
      $child$ ← (TAGNAME,{})
      $V\text{-}AnnMap[child]$ ← $false$ , $C\text{-}AnnMap[child]$ ← $true$
      return ({root, child},{(root, child, '; 'm')}, root)
    else if $e$ is 'TAGNAME' / PathTailExpr then
      $root$ ← ('.',{})
      $child$ ← (TAGNAME,{})
      V ← {root, child}
      E ← {(root, child, '/', 'm') }
      for all $tempQpt$ in GenerateQPT(PathTailExpr) do
        for all (tempQpt.root, n, axis, ann) in tempQpt.E do
          E.add(child, n, axis, ann); V.add(n);
        end for
      end for
      return {(V,E,root)}
    else
      {$e$ is 'TAGNAME' '//' PathTailExpr}
      {Similar to 'TAGNAME' '//' PathTailExpr}
    end if
  else if e istype PredExpr then
    if e is PathExpr then
      return GenerateQPT($PathExpr$)
    else if e is PathExpr Comp Literal then
      $pathset$ = GenerateQPT($PathExpr$)
      for all $pathqpt$ in $pathset$ do
        for all leaf nodes oldnode=(name,pred) in pathqpt do
          $newnode$ = ($name$,pred ∪ {'Comp Literal'})
          $V\text{-}AnnMap[newnode]$ = false,
          $C\text{-}AnnMap[newnode]$ = $C\text{-}AnnMap[oldnode]$
          tempqpt.V.replace(oldnode, newnode)
        end for
      end for
      return $pathset$
    else
      {e is PathExpr1 Comp PathExpr2}
      $pathset1$ = GeneratePST($PathExpr1$)
      $pathset2$ = GeneratePST($PathExpr2$)
      for all leaf node l in $pathset1 \cup pathset2$ do
        $V\text{-}AnnMap[l]$ = true
        $C\text{-}AnnMap[l]$ = false
      end for
      return $pathset1 \cup pathset2$
    end if
  end if
```

Figure 22: Algorithm for producing QPT for PathTailExpr & PredExpr

the given predicate. Further, we indicate that the value of "isbn" is required since it is used in a predicate; and the content of "title" is required since it is part of the view results. Next, we generate the QPT for the path expressions in the for clause, and the resulting QPT is shown in Figure 23(c). Finally, we bind the set of QPTs that generated using the where clause and the return clause to the variable in the for clause. In our example, we simple replace the node "$book" with the leaf node "//book"in Figure 23(c) and Figure 23(d) shows the final QPT. Note that the C-Annotation of the leaf node "//book" in Figure 23(c) is changed to false because the algorithm determines that it is not part of the view results (lines 21 -26 in Figure 24, and other annotations are kept as expected.

C. PDT DEFINITIONS

In this section, we generalize the definitions of PDT described earlier in Section 4 so that it also handles the cases where the root of the QPT is mapped to arbitrary nodes in

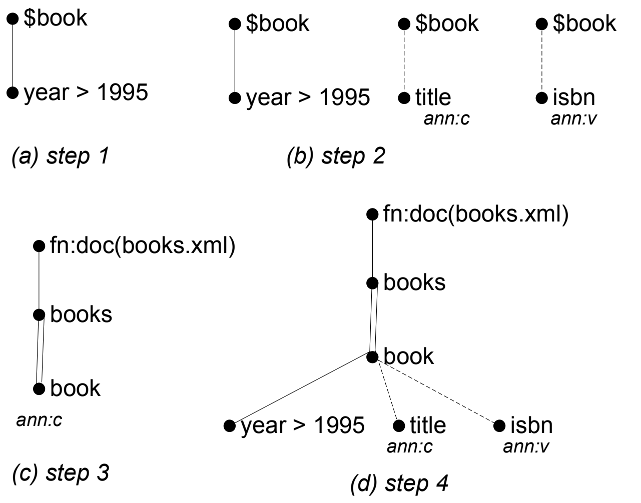


Figure 23: Illustrating the QPT algorithm

an XML database.

We first introduce some notation. We $Nodes(D)$ to denote the set of nodes in an XML database D, $FreeVars(E)$ to denote the set of free variables in a query expression E, $Env(D, FreeVars(E))$ to denote the evaluation enviroment which binds variables in $FreeVars(E)$ to nodes in $Nodes(D)$, $UE(D, FreeVars(E))$ to denote the universe of such enviroments. In $Env(D, FreeVars(E))$, we use $var \Rightarrow n$ to denote that var in $FreeVars(e)$ is bounded to the node n. Similarly, for a QPT Q, we say $Env(D, Q)$ is an enviroment that binds Q.root whose name is a free variable to a node in the database D, and $UE(D, Q)$ is the universe of such enviroments. Note by definition of QPT, only the root of a QPT can be a free variable.

Further, if $QSet = GenerateQPT(E)$ is a set of QPTs corresponding the expression E, then $\forall Q \in QSet$, $\forall \delta \in UE(D, FreeVars(E)), \forall \delta' \in UE(D, Q)$, $(\exists x \in FreeVars(E)$, $x = Q.root.name \Rightarrow \delta'(Q.root) = \delta(x))$. In this case, for notatinonal convenience, we use $UE(D, FreeVars(E))$ and $UE(D, Q)$ interchangeably.

Finally, given a node $d \in Nodes(D)$, we use $T(d)$ to denote the XML sub-tree rooted at d, and $T(d)$ is a 4-tuple $(V, E, Tag, Value)$ where V is the set of nodes in T(d), E is the set of edges in T(d), Tag are the mappings from nodes in V to their tag names, and Value are the mappings from nodes in V to their data values.

Now we generalized the notions of PDTs defined in the main body to handle arbitrary nodes. Since the PDT captures both ancestor and descendant restrictions, for ease of exposition we first define the notion of *candidate elements* that only capture descendant restrictions. Given database D, a QPT Q, an enviroment $\delta \in UE(D, FreeVars(Q))$, $\delta(r_q) = d$ where $d \in Nodes(D)$, the set of candidate element associated with a node $n \in (Q.V - \{Q.root\})$, denoted by $CE(n, d)$, is defined recursively as follows:

- If n is a leaf node in Q, then $CE(n, d) = \{e | e \in T(d).V \wedge Tag(e) = n.name$ $\wedge \forall P \in n.preds(satisfies(e, P))$.
- If n is a non-leaf node in Q, then $CE(n, d) = \{e | e \in T(d).V \wedge Tag(e) = n.name) \wedge \forall nc((n, nc, ' /', ' m') \in Q.E \Rightarrow \exists ec \in CE(nc, d)\ (parent(e, ec))$ $\wedge (n, nc, ' //', ' m') \in Q.E \Rightarrow \exists ed \in CE(nc, d)\ (anc(e, ed)))\}$

```
GenerateQPT (Expr e) : QPTSet

  if e istype FLWORExpr then
    qptSet ← ∅
    {First create QPTS with variable references, then use
    for/let clauses to bind variables}
    if FLOWRExpr.WhereClause is present then
      PredExpr = FLWORExpr.WhereClause.PredExpr
      qptSet←GenerateQPT(PredExpr)
      ∀node ∈ qptSet, C-AnnMap[node] = false
    end if
    RetExpr = FLWORExpr.ReturnClause.RetExpr
    qptSet ← qptSet ∪ GenerateQPT(RetExpr)
    {Process for/let clauses from the inner-most one to
    the outer-most one}
    for all forLetClause in forLetClauses do
      if forLetClause is ForClause then
        VAR ← forLetClause.VAR
        pathSet ←
        GenerateQPT(forLetClause.PathExpr)
        for all pathQpt in pathSet do
          for all prevQpt in qptSet where prevQpt.root
          is VAR do
            pathQpt.E ← pathQpt.E ∪ {$(l, n, axis, ann)$
            $|l \in Leaf(pathQpt)$
            $\wedge(prevQpt.root, n, axis, ann) \in predQpt.E$}
            for all leaf node l in pathQpt do
              if prevQpt corresponds to RetExpr ∧
              prevQpt.V={prevQpt.root} then
                C-AnnMap[l] ←
                C-AnnMap[prevQpt.root]
              else
                C-AnnMap[l] ← false
              end if
            end for
          end for
          qptSet ← qptSet ∪ {pathQpt}
        end for
      else
        {forLetClause is LetClause}
        {Similar to ForClause}
      end if
    end for
    return qptSet

  else
    {e is RetExpr}
    if e is Expr then
      GenerateQPT(Expr)
    else if e is '<' TAGNAME '>' RetExprList '<'
    TAGNAME '>' then
      tempset ← ∅
      for all RetExpr1 in RetExprList do
        currset ← GenerateQPT(RetExpr1)
        for all qpt in currset where qpt.root is VAR do
          E' ← qpt.E with all (qpt.root, n, axis, ann)
          edges replaced with (qpt.root, n, axis, 'o')
        end for
        tempset.add(currSet.V, E', currSet.root)
      end for
    else
      {e is Expr ',' Expr}
      tempset ← ∅
      for all Expr1 in e do
        currset ← GenerateQPT(Expr1)
        for all qpt in currset where qpt.root is VAR do
          E' ← qpt.E with all (qpt.root, n, axis, ann)
          edges replaced with (qpt.root, n, axis, 'o')
        end for
        tempset.add(currSet.V, E', currSet.root)
      end for
    end if
  end if
  return QPTSet
```

Figure 24: Algorithm for producing QPT: FLWOR-Expr

Next we define notions of *PDT elements* which capture both ancestor and descendant restrictions. Given an XML document D, a QPT Q, the set of PDT elements associated with a node $n \in Q$, denoted by *PE(n, D)*, is defined recursively as follows:

- If n is the root node of Q, then PE(n, d) = CE(n, d)
- If n is the non-root node in Q and np is the parent node of n in Q, then *PE(n, d) = {e|e ∈ CE(n, d) ∧ (((np, n, '/', ann) ∈ Q ⇒ ∃pe ∈ PE(np, d), parent(pe, e)) ∧ ((np, n, '//', ann) ∈ Q ⇒ ∃pe ∈ PE(np, d), anc(pe, e))) }*

Given a node d, a QPT Q, and an enviroment δ s.t. $\delta(Q.root) = d$, and a set of keywords K, we say $PDT(Q, \delta, K)$ is the *minimal* 5-tuple $I = (V, E, Tag, Val, Cont)$ that satisfies the following properties:

- I.V = $\cup_{q \in Q}$ PE(q, d) (nodes of the PDT is the union of pdt elements with respect to all QPT nodes.)
- $\forall n \in Q \forall e \in I$ (e ∈ PE(n, d) ∧ *n.v-ann* = $'v'$ ⇒ *(e, Val(e)) ∈ I.Val)* (all elements corresponding to 'v' nodes have a value)
- $\forall n \in Q \forall e \in I$ (e ∈ PE(n, d) ∧ *n.c-ann* = $'c'$ ⇒ *(e,(id(e),len(e),{(k,tf(e,k)) |k ∈ K })) ∈ I.Cont)* (all elements corresponding to 'c' nodes have the id, length and tf values of the node).

D. CORRECTNESS OF THE ALGORITHMS

In this section we prove the correctness of our QPT and PDT generation algorithms. We first prove that for a query expression E which conforms to the core XQuery grammar, $GenerateQPT(E)$ generates the correct set of QPTs (Theorem D.1). We then show that $GenerateQPT(E) = GenerateQPT(E')$ where E is a query expression that conform to our grammar and E' is the corresponding normalized query in the core grammar (Theorem D.6). Finally we prove that given a single QPT q, $GeneratePDT(q)$ produces the correct PDT as per the definition (Theorem F.1).

D.1 Correctness of GenerateQPT

We first introduce some notation. We use the XQuery formal semantics for evaluating queries, and we use $Env \vdash E \Rightarrow V$ to denote that in the evaluation context Env, the query expression E evaluates to the value V. For notational convenience, we also use $Eval(E, Env)$ to denote V. Note that Env captures both static context and dynamic context used in the formal semantics.

Further, in our post-processing, we say I is the function transforming Dewey IDs to node contents in the database, PDTTF is the tf calculation function, and PDTByteLength is the byte length calculation function, and len(e) is the byte length of a materialized element e, then we can prove the correctness of GenerateQPT in Theorem D.1

Theorem D.1 (Correctness of GenerateQPT). *Given a set of keywords KW, an XQuery query expression E that conforms to the core grammar, a database instance D, then* $\forall \delta \in UE(D, FreeVars(E))$

(a) I(Eval(E, {Q.root ⇒ PDT(Q, KW, δ) | Q ∈ GenerateQPT(E)})) = Eval(E, δ) (The result sequences, after being transformed, are identical)

(b) ∀e ∈ Eval(E, {Q.root ⇒ PDT(Q, KW, δ) | Q ∈ GenerateQPT(E)}), e′ ∈ Eval(E, δ), I(e) = e′ ⇒ PDTByteLength(e) = len(e′) (The byte lengths of each element are identical)

(c) $\forall e \in$ *Eval(E,* $\{Q.root \Rightarrow PDT(Q, KW, \delta) \mid Q \in$ *GenerateQPT(E)*$\}$), $e' \in$ *Eval(E,* δ), $I(e) = e' \Rightarrow (\forall k \in KW,$ *PDTTF(e, k) = tf(e′,k)) (The term frequencies of each keyword in each element is identical)*

Before proving the lemma, we first prove two supporting lemmas.

If $Leaf(Q)$ is the set of the leaf nodes in a QPT Q, then we first show Lemma D.2 showing that GenerateQPT(PathExpr) is a singleton set and it has only one leaf node.

Lemma D.2 (GenerateQPT(PathExpr)). *Given a path expression E,*

- $|GenerateQPT(E)|=1$, *and*
- $\forall Q \in GenerateQPT(E)$, $|Leaf(Q)| = 1$.

Proof. *Sketch* We show the lemma by structural induction on E.

Base case: E = fn:doc(Name) or VAR or '.' By line 7 in Figure 21, it is easy to see that the base case holds.

Inducting hypothesis: Suppose Lemma D.2 holds for subexpressions of E, now we show it holds for E.

There are several cases and their proofs are similar. Now we only show for E = fn:doc(Name) '/' PathTailExpr.

First, by I.H., we know that |GenerateQPT(PathTailExpr)| = 1 and $|Leaf(Q')| = 1$ where $\{Q'\}$ = GenerateQPT (PathTailExpr). We also know that |GenerateQPT(fn:doc(Name))| = 1 and $|Leaf(Q'')| = 1$ where $\{Q''\}$ = GenerateQPT(fn : doc(Name)).

Then by lines 11-16 in Figure 21, we know that GenerateQPT(E) = {Q } where $Q.V = Q''.V \cup Q'.V - \{Q'.root\}$, $Q.E = Q''.E - \{(Q''.root, x, axis, ann) | x \in Q''.V\}$ $\cup\{(Q'.root, x, axis, ann) | (Q''.root, x, axis, ann) \in Q''.E\}$, and $Leaf(Q) = Leaf(Q')$. Hence $|GenerateQPT(E)| = 1$ and $\forall Q \in GenerateQPT(E), |Leaf(Q) = 1|$

□

Lemma D.3 (Mandatory Child Edges). *Given a query expression E, an XML database D,* $(\forall \delta \in UE(D, FreeVars(E)),$ $\exists Q \in GenerateQPT(E), c \in Q.V, r \in Nodes(D)$ $(\delta(Q.root) = r \wedge (Q.root, c,' /',' m') \in Q.E \wedge (\nexists n \in CE(c, r), parent(r, n)))$ $\Rightarrow Eval(E, \delta) = ()$.

Proof. *Sketch* We prove Lemma D.3 by structural inductions on E.

Base case 1: E = fn:doc(Name) or VAR or '.' This case is handled by lines 5- 7 in Figure 21. It is easy to see that $E_q = \emptyset$. Therefore the lemma is vacously true.

Base case 2: E = TAGNAME This case is handled by lines 3-8 in Figure 22. It is easy to see that GenerateQPT(E) is a singleton set. Assume $\{Q\}$ $\in GenerateQPT(E)\}$, then by the algorithm we know $Q.V = \{Q.root, l\}$ where l is the leaf node and $l.name = TAGNAME$.

By definition, if n is not in $CE(l, r)$, then $n.Name \neq TAGNAME$. Therefore if $(\nexists n \in CE(l, r)$ $parent(r, n))$, we know that r does not have a child node with the tag name TAGNAME.

On the other hand, by the semantics of E, $Eval(TAGNAME, \delta)$ is evaluated by invoking *NameTest* on child nodes of r. Since r does not have a child node with the tag name TAGNAME, we can infer that $Eval(TAGNAME, \delta) = ()$.

Induction hypothesis: Suppose the lemma holds for subexpressions of E. We now show the lemma also holds for E.

Here we only prove the case *E = for VAR in PathExpr return Expr* as it covers the main points of all other cases.

Case 1: E = for VAR in PathExpr return Expr This case is handled by lines 12-26 in Figure 24. Essentially, we first obtain the set of QPTs corresponding to *Expr*, then if VAR is referenced in *Expr* (as the root), we bind the VAR node in *Expr* to the leaf nodes of GenerateQPT(PathExpr).

By Lemma D.2, we know that $|GenerateQPT(PathExpr)| = 1$. Assume $\{P\} = GenerateQPT(PathExpr)$, and w.o.l.g., we assume $\{X\} = GenerateQPT(Expr)$.

Now we assume $\exists Q \in GenerateQPT(E), \exists c \in Q.V$ $((r_q, c,' /',' m') \in Q.E \wedge \delta(Q.root) = r$ $\wedge(\nexists n \in CE(c, r)\ parent(r, n))$, and need to show that $Eval(E, \delta) = ()$.

There are two cases depending on the value of $X.root$.

Case A: $Q.root \neq VAR$. In this case, Expr does not reference VAR. Therefore by lines 19-26 in Figure 24, we know that GenerateQPT(E) = $\{P, X\}$. I.e., $Q = P$ or $Q = X$. If $Q = P$, then by I.H., $Eval(P, \delta) = ()$. According to the semantics of E, we know that $Eval(E, \delta) = ()$. Otherwise $Eval(P, \delta) \neq ()$ and by I.H., $Eval(X, \delta) = ()$. Again, according to the semantics, $Eval(E, \delta) = Eval(X, \delta)$ and hence $Eval(E, \delta) = ()$.

Case B: $Q.root = VAR$. In this case, Expr does reference VAR. By lines 19-26 in Figure 24, we know that GenerateQPT(E) = $\{P'\}$ where $P'.V = P.V \cup X.V - \{X.root\}$, $P'.E = P.E \cup X.E - \{(X.root, l, axis, ann)\}$ $\cup\{(l_p, l, axis, ann)\ |(X.root, l, axis, ann) \in X.E\}$ where l_p is the leaf node in P, and $P'.root = P.root$.

Now if $l_p = P.root$, i.e., if P is a tree with a single node, then $P = X$ and therefore we can apply I.H. on X and conclude that $Eval(E, \delta) = ()$.

Otherwise $l_p \neq P.root$. Since $P'.root = P.root$, hence $c \in P$. Now we reason by analyzing the relations of $CE(c, r)$ w.r.t. P and P', denoted by $CE_P(c, r)$ and $CE_{P'}(c, r)$, respectively.

First, by definition of candidate elements, we know $CE_{P'}(c, r) \subseteq CE_P(c, r)$ since intuitively P' contains all edges in P and has additional edges in X. Therefore $\nexists n \in CE_{P'}(c, r)$ $parent(r, n)$ implies that $\nexists n \in CE_P(c, r)\ parent(r, n)$. Therefore we can apply I.H. on PathExpr and conclude that Eval (PathExpr , δ)=(), and hence according to the semantics of E, $Eval(E, \delta) = ()$.

Hence the lemma holds for E. □

We can similarly show the following theorem.

Lemma D.4 (Mandatory Descendant Edges). *Given a query expression E, an XML database D,* $(\forall \delta \in UE(D, FreeVars(E)),\ \exists Q \in GenerateQPT(E), c \in Q.V, r \in Nodes(D)\ ((Q.root, c,' //',' m') \in Q.E \wedge \delta(Q.root) = r \wedge(\nexists n \in CE(c, r)\ parent(r, n))) \Rightarrow Eval(E, \delta) = ()$.

Lemma D.3 and Lemma D.4 indicate that if an element corresponding to the root of an expression (and its QPT) does not have a mandatory child (descendant), then the evaluation results using this element as the context is an empty sequence.

Now we show Theorem D.1(a).

Proof. We prove Theorem D.1(a) by structural induction on the query expression. For notational convenience, let $\delta' = \{Q.root \Rightarrow PDT(Q, KW, \delta) |\ Q \in GenerateQPT(E)\}$.

Base case 1: E = fn:doc(Name) or VAR or '.'

This case is handled by lines 3-7 in Figure 21. By line 7, we know that GenerateQPT(E)={Q} where $Q.V = \{Q.root\}$, $Q.E = \emptyset$ and $Q.root$.name = E. If $\delta(Q.root) = d$, then by the formal semantics, we know that $Eval(E, \delta) = d$. On the other hand, since $Q.root$.name=E, by definition of δ and the formal semantics, we know that $Eval(E, \{Q.root \Rightarrow PDT(Q, KW, \delta)\}) = PDT(Q, KW, \delta).root$. Since $\delta(Q.root) = d$, by the definition of PDT, $PDT(q, KW, \delta).root = d$, and therefore $Eval(E, \{Q.root \Rightarrow PDT(Q.root, KW, \delta)\}) = d$. Last, by line 6 in Figure 21, we know that *C-AnnMap[Q.root] = true*. Therefore it is easy to see I(Eval(E, $\{r_q \Rightarrow$ PDT(q, KW, δ)$|$ $Q \in GenerateQPT(E)\} = Eval(E, \delta)$.

Thus the base case 1 holds.

Base case 2: E = TAGNAME

This case is handled by lines 3-8 in Figure 22. It is easy to see that GenerateQPT(E) is a singleton set. Assume $\{Q\}$ $\in GenerateQPT(E)\}$, then by the algorithm we know $Q.V$ $= \{Q.root, l\}$ where l is the leaf node, $l.name = TAGNAME$, and $Q.root.name =' .'$. $Q.E = \{(Q.root, l, '/', 'm')\}$

If $\delta(Q.root) = e$ where $e \in D'.V$ where D' is a document in D, $NameTest(S, tag) = V$ is the standard function in the specification of formal semantics that given a sequence of nodes S, a tag name tag, returns the sequence of nodes $V \subseteq S$ and $\forall node \in V\ node.name = tag$. Then according to the formal semantics, E=TAGNAME is evaluated using $NameTest(Value('.').Children, TAGNAME)$ where $Value('.')$ is the node '.' is bounded to.

Let $C = \{c \in e.Children | Tag(c) = TAGNAME\}$, then $Eval(E, \delta) = Concatenate(C)$ where Concatenate(C) concatenates items in the set C in document orer. On the other hand, since $Q.E = \{(Q.root, l, '/', 'm')\}$, by definition of PDT, we know that $\forall c \in C$, $c \in CE(l, D') \wedge c \in PDT(Q, KW, \delta).V$. Hence $Eval(E, \delta') = Concatenate(C)$. Finally, since *C-AnnMap[l]*=true (line 7 in Figure 22), we know that $\forall x \in Eval(E, \delta')$, $x has\ id$. It is then easy to see that $I(Eval(E, \delta')) = Eval(E, \delta)$.

Thus the base case 2 holds.

Induction Hypothesis: For an expression E that is derived using grammar rules, suppose Theorem D.1 holds for its sub-expressions.

We will now show that Theorem D.1 holds for E itself. There are six cases, one for each different kind of derivation.

Case 1: E= for Var in PathExpr return Expr

The main evaluation rules of $Eval(E, \delta)$ are as follows.

The iteration expression $PathExpr$ is evaluated to produce the sequence $Item_1$, ..., $Item_n$. For each item $Item_i$ in this sequence, the body of the for expression $Expr$ is evaluated in the environment δ extended with Var bound to $Item_i$. This produces values $Value_i$, ..., $Value_n$ which are concatenated to produce the result sequence.

The specific rules for $Eval(E, \delta)$ are:

$$\frac{\begin{array}{c}\delta \vdash PathExpr \Rightarrow Item_1, ..., Item_n \\ \delta + VAR \Rightarrow Item_1 \vdash Expr \Rightarrow Value_1 \\ \cdots \\ \delta + VAR \Rightarrow Item_n \vdash Expr \Rightarrow Value_n\end{array}}{\delta \vdash E \Rightarrow Value_1, ..., Value_n}$$

The evaluation rules for $Eval(E, \delta')$ are:

$$\frac{\begin{array}{c}\delta' \vdash PathExpr \Rightarrow Item'_1, ..., Item'_m \\ \delta' + VAR \Rightarrow Item'_1 \vdash Expr \Rightarrow Value'_1 \\ \cdots \\ \delta' + VAR \Rightarrow Item'_m \vdash Expr \Rightarrow Value'_m\end{array}}{\delta' \vdash E \Rightarrow Value'_1, ..., Value'_m}$$

W.o.l.g, assume GenerateQPT(Expr) = $\{Q_e\}$. There are two cases according to the value of $Q_e.root.name$.

Case A: $Q_e.root.name \neq VAR$. Intuitively, in this case, $Expr$ does not reference VAR. Therefore by $\delta + VAR \Rightarrow Item_i \vdash Expr \Rightarrow Value_i$, we can infer that $\delta \vdash Expr \Rightarrow Value_i$. This indicates that $\forall i, j, Value_i = Value_j = Value$. Therefore $Eval(E, \delta) = (Value, ..., Value)$. Similarly, we can infer that $\delta' \vdash Expr \Rightarrow Value'_i \wedge \forall i, j, Value'_i = Value'_j = Value' \wedge Eval(E, \delta') = (Value', ..., Value')$ Further, since $r_e \neq VAR$, by lines 19-26 and line 29 in Figure 24, we know that $Q_e \in GenerateQPT(E)$ and therefore $Q_e.root \Rightarrow PDT(Q_e, KW, \delta) \in \delta'$. Therefore by I.H. on the sub-expression $Expr$, we know that $Value = I(Value')$. Further, by I.H. on PathExpr, we know that $Eval(PathExpr, \delta)$ $= I(Eval(PathExpr, \delta'))$ and hence $m = n$. Therefore we finally know $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case B: $Q_e.root.name = VAR$. There are two different cases depending on whether $Q_e.root$ has child edges in Q_e.

Case B.1: $Q_e.root$ has no child edges. In this case, the return expression $Expr$ is just VAR. By lines 19-26 in Figure 24, we know that $Q_e \notin GenerateQPT(E)$. Therefore GenerateQPT(E) = GenerateQPT(PathExpr). Hence if $\delta'' = \{Q_q.root \Rightarrow PDT(Q_q, KW, \delta) |\ Q_q \in$ GenerateQPT (PathExpr)$\}$, then $\delta' = \delta''$. So we can apply I.H. on PathExpr and know that $I(Item'_i) = Item_i$.

Then since $Q_e.V = \{Q_e.root\}$, $Eval(Expr, \delta$ $+VAR \Rightarrow Item_i) = Item_i$ and $Eval(Expr, \delta + VAR \Rightarrow Item'_i) = Item'_i$. Hence $Eval(Expr, \delta + VAR \Rightarrow Item_i)$ $= I(Eval(Expr, \delta{+}VAR \Rightarrow Item'_i))$ for all i. Consequently, $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case B.2: $Q_e.root$ has child edges. In this case, by lines 19-26 in Figure 24, the algorithm will create edges between the leaf nodes in GenerateQPT(PathExpr) and the child nodes of $Q_e.root$. W.o.l.g, assume $Q_e.root$ has a single child x. By Lemma D.2, we know that $|GenerateQPT(PathExpr)|$=1. Let $\{q\} = GenerateQPT(PathExpr)$. Then by Lemma D.2, we know that $|Leaf(q) = 1|$. Let $l \in q.V$ be the single leaf node in q. Then by lines 19-26 in Figure 24, it is easy to see that $|GenerateQPT(E)| = 1$. Assume $\{Q_q\} = GenerateQPT(E)$. Now depending on the edge annotations, there are further two different cases. Let $l' \in Q_q.V$ be l in GenerateQPT(E), and $e = (l, x, axis, ann) \in q.E$.

First, if $ann =' o'$, then by definition, $CE(l) = CE(l')$. Therefore by lines 19-26 in Figure 24, it is easy to see that $PDT(q, KW, \delta).V = PDT(Q_q, KW, \delta).V - \{x\}$ and PDT(q, KW, δ).E = PDT(Q_q, KW, δ).E-{e}). Hence Eval(PathExpr, δ') = Eval(PathExpr,$\{Q_q$.root$\Rightarrow$ PDT(q, KW, δ)$\}$). Further, by I.H. on $PathExpr$, we know that $I(Eval(PathExpr,$ $\{Q_q.root \Rightarrow PDT(q, KW, \delta)\})) = Eval(PathExpr, \delta)$, and therefore we have $I(Eval(PathExpr, \delta')) = Eval(PathExpr, \delta)$. I.e., for all i, $I(Item'_i) = Item_i$.

Then by I.H. on $Expr$, we know that $\forall \delta\ I(Eval(Expr,$ $\{Q_q.root \Rightarrow PDT(q, KW, \delta) |\ q =\in GenerateQPT(Expr)\}))$ $= Eval(Expr, \delta)$. If $\delta'' = \delta + VAR \Rightarrow Item_i$, then since $I(Item'_i) = Item_i$, it is easy to see that $(Q_e.root \Rightarrow Item'_i) = \{Q_q.root \Rightarrow PDT(q, KW, \delta'') |\ q \in GenerateQPT(Expr)\}$ and hence $I(Eval,$
$\{Q_q.root \Rightarrow PDT(q, KW, \delta'') |\ q \in GenerateQPT(Expr)\}))$

$= Eval(Expr, \delta'')$. Therefore $Value_i = I(Value_i')$ for all i. Consequently, $I(Eval(E, \delta')) = Eval(E, \delta)$.

Second, if $ann = 'm'$. If Eval(PathExpr, δ') = Eval(PathExpr, $\{Q_q$.root$\Rightarrow$ PDT(q, KW, δ) | q $\in$ GenerateQPT(E)$\}$) (i.e., $I(Item_i') = Item_i$ for all i), then we can use the same argument as above. Otherwise we know that $Eval(PathExpr, \delta')$ $\subset$ Eval(PathExpr,$\{Q_q$.root $\Rightarrow$ PDT(q, KW, δ) $|q \in$ GenerateQPT(E)$\}$). Let $X = Eval(PathExpr, \delta') \cap Eval(PathExpr,$ $\{Q_q.root \Rightarrow PDT(q, KW, \delta)\ |q \in GenerateQPT(E)\})$ and $Y = Eval(PathExpr,$ $\{Q_q.root \Rightarrow PDT(q, KW, \delta)\ |q \in GenerateQPT(E)\})$ $- Eval(PathExpr, \delta')$. For $Item_i'$ in X, we can use the similar argument in the Case B.1 and show that $I(Value_i') = Value_i$. Further, by the definition of PDT and definitions of CE, we know that $\exists c, (l, c, axis, 'm') \in Q_q.E$, $\forall y \in Y$, $\nexists n \in CE(n, D) parent(y, n)$. Then, by Lemma D.3, we can infer that $\forall y \in Y$, $Eval(Expr, Q_e.root \Rightarrow y) = ()$. Then we can use I.H. on $Expr$ and infer that $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case 2: E = for VAR in PathExpr return $\langle$TAGNAME$\rangle$ Expr $\langle$/TAGNAME$\rangle$

The evaluation rules are similar to Case 1 with the following additional rule for constructing the element.

$Eval(\langle TAGNAME \rangle Expr \langle / TAGNAME \rangle, \delta)$
$= element\ QNameEval(Expr, \delta)$, and
$Eval(\langle TAGNAME \rangle Expr \langle / TAGNAME \rangle, \delta')$
$= element\ QName\{Eval(Expr, \delta')\}$,
where *element QName* is the element construction function defined in the formal semantics.

Now we present the entire rules of $Eval(E, \delta)$.

$$\frac{\begin{array}{c}\delta \vdash PathExpr \Rightarrow Item_1, ..., Item_n \\ \delta + VAR \Rightarrow Item_1 \vdash Expr \Rightarrow Value_1 \\ ... \\ \delta + VAR \Rightarrow Item_n \vdash Expr \Rightarrow Value_n\end{array}}{\delta \vdash E \Rightarrow elementQName\{Value_1\}, ..., elementQName\{Value_n\}}$$

The evaluation rule for $Eval(expr, \delta')$ is:

$$\frac{\begin{array}{c}\delta' \vdash PathExpr \Rightarrow Item_1', ..., Item_n' \\ \delta' + VAR \Rightarrow Item_1' \vdash Expr \Rightarrow Value_1' \\ ... \\ \delta' + VAR \Rightarrow Item_n' \vdash Expr \Rightarrow Value_n'\end{array}}{\delta' \vdash E \Rightarrow elementQName\{Value_1'\}, ..., elementQName\{Value_n'\}}$$

By Lemma D.2, we know $|GenerateQPT(PathExpr)| = 1$. Let $\{q\} = GenerateQPT(PathExpr)$, and w.o.l.g, assume GenerateQPT(Expr) = $\{Q_e\}$. Similar to Case 1, there are two cases according to the value of $Q_e.root$.

Case A: $Q_e.root \neq VAR$. The proof of this case is identical to Case A in proofs of Case 1 and therefore the proof is skipped here.

Case B: $Q_e.root = VAR$. There are two different cases depending on whether $Q_e.root$ has child edges.

Case B.1: $Q_e.root$ has no child edges. The proof of this case is identical to Case B.1 in Case 1 except that instead of returning $Value_i$, $Eval(E, \delta)$ now returns sequence of *element QName* $\{Value_i\}$, and $Eval(E, \delta')$ now returns sequence of *element QName* $\{Value_i'\}$. Therefore the proof is skipped here.

Case B.2: $Q_e.root$ has child edges. In this case, lines 19-26 in Figure 24, the algorithm will create edges between the leaf nodes in GenerateQPT(PathExpr) and the child nodes of r_e. W.o.l.g, assume $Q_e.root$ has a single child x. Assume l is the single leaf node in q. Assume l' is l in GenerateQPT(E). Then by lines 20 and 47 in Figure 24, we know that $\forall e = (l, x, axis, ann)$, $ann = 'o'$.

Therefore we can first use the same argument as in Case B.2 in Case 1 when ann='o' and infer that for all i, $I(Item_i') = Item_i$. Then we can also use the same argument as in Case B.2 in Case 1 and use I.H. on $Expr$ to infer that $Value_i = I(Value_i')$ for all i. Hence $element\ QName\{Value_i\}$ $= I(element\ QName\{Value_i'\})$ for all i. Consequently, $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case 3: E = for VAR in PathExpr return Expr1,Expr2

Let $Expr' = Expr1, Expr2$, the evaluation rules of $Eval(Expr1', \delta)$ is,

$$\frac{\begin{array}{c}\delta \vdash Expr1 \Rightarrow Value_1 \\ \delta \vdash Expr2 \Rightarrow Value_2\end{array}}{\delta \vdash Expr' \Rightarrow Value_1, Value_2}$$

And the complete rules of $Eval(E, \delta)$ are,

$$\frac{\begin{array}{c}\delta \vdash PathExpr \Rightarrow Item_1, ..., Item_n \\ \delta + VAR \Rightarrow Item_1 \vdash Expr1 \Rightarrow Value_{11} \\ \delta + VAR \Rightarrow Item_1 \vdash Expr2 \Rightarrow Value_{12} \\ ... \\ \delta + VAR \Rightarrow Item_n \vdash Expr1 \Rightarrow Value_{n1} \\ \delta + VAR \Rightarrow Item_n \vdash Expr2 \Rightarrow Value_{n2}\end{array}}{\delta \vdash E \Rightarrow Value_{11}, Value_{12}, ..., Value_{n1}, Value_{n2}}$$

The evaluation rule for $Eval(expr, \delta')$ is:

$$\frac{\begin{array}{c}\delta' \vdash PathExpr \Rightarrow Item_1', ..., Item_n' \\ \delta' + VAR \Rightarrow Item_1' \vdash Expr1 \Rightarrow Value_{11}' \\ \delta' + VAR \Rightarrow Item_1' \vdash Expr2 \Rightarrow Value_{12}' \\ ... \\ \delta' + VAR \Rightarrow Item_n' \vdash Expr1 \Rightarrow Value_{n1}' \\ \delta' + VAR \Rightarrow Item_n' \vdash Expr2 \Rightarrow Value_{n2}'\end{array}}{\delta' \vdash E \Rightarrow Value_{11}', Value_{12}, ..., Value_{n1}, Value_{n2}}$$

Therefore we need to show that (1) $\forall i, I(Value_{i1}') = Value_{i1}$, and (2) $I(Value_{i2}') = Value_{i2}$.

We now prove (1) holds and it is analogous to prove (2). By Lemma D.2, we know $|GenerateQPT(PathExpr)| = 1$. Let $\{q\} = GenerateQPT(PathExpr)$, and $\forall Q_e \in$ GenerateQPT(Expr1, δ), by line 57, we know that an optional edge will be created between leaf nodes of q and the child nodes of $Q_e.root$. Now similar to Case 2, there are two difference cases.

Case A: $Q_e.root \neq VAR$. The proof of this case is identical to Case A in Case 1 and therefore the proof is skipped here.

Case B: $Q_e.root = VAR$. There are two different cases depending on whether r_e has child edges.

Case B.1: $Q_e.root$ has no child edges. The proof of this case is identical to Case B.1 in Case 1 and therefore the proof is skipped here.

Case B.2: $Q_e.root$ has child edges. The proof of this case is similar to Case B.2 in Case 2 in the sense that optional edges are created between leaf nodes in q and the child nodes of $Q_e.root$. Therefore we can show that $I(Eval(PathExpr, \delta'))$

$= Eval(PathExpr, \{r_q \Rightarrow PDT(q, KW, \delta)\}$, and therefore $I(Item'_i1) = Item_i1$, and we can also similarly infer that for all i, $I(Value'_{i2}) = Value_{i2}$.

Consequently, $I(Eval(E, \delta')) = Eval(E, \delta)$

Case 4: E= let Var := PathExpr return Expr

The evluation rule of $Eval(E, \delta)$ is,

$$\frac{\delta \vdash PathExpr \Rightarrow Item \qquad \delta + VAR \Rightarrow Item \vdash Expr \Rightarrow Value}{\delta \vdash E \Rightarrow Value}$$

The evluation rule of $Eval(E, \delta')$ is,

$$\frac{\delta' \vdash PathExpr \Rightarrow Item' \qquad \delta' + VAR \Rightarrow Item' \vdash Expr \Rightarrow Value'}{\delta' \vdash E \Rightarrow Value'}$$

By line 33 in Figure 24, the algorithm handles this case the same way as Case 1, and the proof of this case can be viewed as a special case of Case 1 in which $n = 1$. Therefore the complete proof is skipped.

And we can similarly prove the cases of *E= let Var := PathExpr return* $\langle$*TAGNAME*$\rangle$*Expr*$\langle$*/TAGNAME*$\rangle$ and *E= let Var := PathExpr return Expr1,Expr2*.

Case 5: E = 'if' Expr1 'then' Expr2 'else' Expr3

The evaluation rules for $Eval(E, \delta)$ is,

$$\frac{\delta \vdash fn : boolean(Expr1) \Rightarrow true \qquad \delta \vdash Expr2 \Rightarrow Value_1}{\delta \vdash E \Rightarrow Value_1}$$

and

$$\frac{\delta \vdash fn : boolean(Expr1) \Rightarrow false \qquad \delta \vdash Expr3 \Rightarrow Value_2}{\delta \vdash E \Rightarrow Value_2}$$

The evaluation rules for $Eval(E, \delta')$ is,

$$\frac{\delta' \vdash fn : boolean(Expr1) \Rightarrow true \qquad \delta' \vdash Expr2 \Rightarrow Value'_1}{\delta' \vdash E \Rightarrow Value'_1}$$

and

$$\frac{\delta' \vdash fn : boolean(Expr1) \Rightarrow false \qquad \delta' \vdash Expr3 \Rightarrow Value'_2}{\delta' \vdash E \Rightarrow Value'_2}$$

By line 40 in Figure 21, if $Q1 = GenerateQPT(Expr1)$, $Q2 = GenerateQPT(Expr2)$, and $Q3 = GenerateQPT(Expr3)$, then GenerateQPT(E) = Q = $Q1 \cup Q2 \cup Q3$. If $\delta_1 = \{r'_q \Rightarrow PDT(Q', KW, \delta) | Q' \in Q1\}$, $\delta_2 = \{r''_q \Rightarrow PDT(Q'', KW, \delta) | Q'' \in Q2\}$, $\delta_3 = \{r'''_q \Rightarrow PDT(Q''', KW, \delta) | Q''' \in Q3\}$, by definition of PDT, we have $\delta' = \delta_1 \cup \delta_2 \cup \delta_3$.

By I.H., we know that $I(Eval(Expr1, \delta_1)) = Eval(Expr1, \delta)$, $I(Eval(Expr2, \delta_2)) = Eval(Expr2, \delta)$, and $I(Eval(Expr3, \delta_3)) = Eval(Expr3, \delta)$.

Hence we have $I(Eval(Expr1, \delta')) = Eval(Expr1, \delta)$, $I(Eval(Expr2, \delta')) = Eval(Expr2, \delta)$, and $I(Eval(Expr3, \delta'))$ $= Eval(Expr3, \delta)$.

And then it is easy to see that $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case 6: E = QName "(" PathExpr1,..., PathExprn ")"

This case corresponds to function call and the evaluation rules for $Eval(E, \delta)$ is,

$$\frac{\begin{array}{c}\delta \vdash \textit{QName expands to QName(VAR1,...,VARn)}\{\textit{Expr}\} \\ \delta \vdash PathExpr1 \Rightarrow Value_1 \\ \ldots \\ \delta \vdash PathExprn \Rightarrow Value_n \\ \delta + VAR1 \Rightarrow Value_1 + ... + VARn \Rightarrow Value_n \vdash Expr \Rightarrow Value\end{array}}{\delta \vdash E \Rightarrow Value}$$

The rules for $Eval(E, \delta')$ is,

$$\frac{\begin{array}{c}\delta' \vdash \textit{QName expands to QName(VAR1,...,VARn)}\{\textit{Expr}\} \\ \delta' \vdash PathExpr1 \Rightarrow Value'_1 \\ \ldots \\ \delta' \vdash PathExprn \Rightarrow Value'_n \\ \delta' + VAR1 \Rightarrow Value'_1 + ... + VARn \Rightarrow Value'_n \vdash Expr \Rightarrow Value'\end{array}}{\delta' \vdash E \Rightarrow Value'}$$

There are two cases based on whether the function takes parameters.

Case 1: n=0. In this case, the function takes no parameters. By lines 48-60 in Figure 21, GenerateQPT(E) = GenerateQPT(Expr). Further, by I.H., we know that $I(Eval(Expr, \{Q.root \Rightarrow PDT(Q, KW, \delta) \, | Q \in GenerateQPT(Expr)\}\})$ $= Eval(Expr, \delta)$. Hence $I(Eval(Expr, \{Q.root \Rightarrow PDT(Q, KW, \delta) \, | Q \in GenerateQPT(E)\}\}) = Eval(Expr, \delta)$. I.e., $I(Value') = Value$. Therefore $I(Eval(E, \delta')) = Eval(E, \delta)$.

Case 1: $n > 0$. By the evaluation rules and lines 48 - 60 in Figure 24, this case is similar to the case where E *= let VAR1 := PathExpr1 ... let VARn := Exprn return Expr*, which will be shown to be correct (by Case 2 and Theorem D.6). Thereofer the details are skipped here.

□

We now briefly show that Theorem D.1(b) hold. First, for an expression E and an enviroment δ, for an element $e \in Eval(E, \delta)$, $PDTByteLength(e) = \Sigma e'.Length$ where $e' \in e.Descendants \wedge e'$ *is a base element*. Second, note in the algorithm, we set the annotation for the QPT node that is used in constructing the views in C-AnnMap to be true (Theorem D.5) and therefore the required byte lengths of the base elements will be correctly collected and generated in the PDT (Theorem D.5). Therefore if $I(e) = e'' wheree'' \in Eval(E, \delta)$ (Theorem D.1(a), then we know e contains all required base elements and therefore $\Sigma e'.Length = len(e'')$.

If $Nodes(e, D)$ is the set of nodes in the subtree in D rooted at the node e, then we can show the following theorem.

THEOREM D.5 (C-ANNMAP). *Given a query expression E, an XML document D, an enviroment* $\delta \in UE(D, FreeVars(E))$, $\forall e' \in \{Nodes(e, D) | \, e \in Eval(E, \{Q.root \Rightarrow PDT(Q, KW, \delta) \, | Q \in GeneraetQPT(E)\})\}$, $(\exists c \in \{Q.V | Q \in GeneraetQPT(E)\}$ $e' \in CE(c)) \Rightarrow C - AnnMap[c] = true$

PROOF. *Sketch* We prove Theorem D.1(b) by structural inductions on E. Let $\delta' = \{Q.root \Rightarrow PDT(Q, KW, \delta) \, | Q \in GeneraetQPT(E)\}$

Base case: E = fn:doc(Name) or VAR or '.'

In this case, by the algorithm we know GenerateQPT(E) produces a singleton set $\{Q\}$. And by line 6 in Figure 21, *C-AnnMap[Q.root]*=true.

On the other hand, according to formal semantics, we know that $Eval(E, \delta') = r$ where $\delta(Q.root) = r$. Therefore $r \in candidatElems(Q.root)$. Since we just show *C-AnnMap[Q.root]*=true, hence our theorem holds.

Induction hypothesis: Assume the theorem holds for sub-expressions of E. We need to show it holds for E itself.

Here we show the case E = for VAR in PathExpr return Expr to illustrate the main points. Other cases are similar and their proofs are ignored.

Case 1: E = for VAR in PathExpr return Expr

First, by the formal semantics, essentially $Eval(E, \delta') = \{Eval(Expr, \delta'{+}VAR \Rightarrow Item\,|Item \in Eval(PathExpr, \delta')\}$ where we overload the set operator '{}' to concatenate the items in the set. By I.H. on $Expr$, we know that if $e \in Eval(Expr, \delta')$ and $e \in CE(c')$ where c' is a QPT node GenerateQPT(Expr), then C-AnnMap[c'] = true.

Then, by lines 21-26 in Figure 24, we know that for all non-leaf nodes x in GenerateQPT(Expr), C-AnnMap[x] remains the same. Now w.o.l.g., assume {G} = GenerateQPT(Expr). If C-AnnMap[$G.root$] = true and $e \in$ CE(G.root) and $e \in Eval(Expr, \delta')$, then by the formal semantics of XQuery, we know $Item \in Eval(Expr, \delta')$ where $Item \in Eval(PathExpr, \delta')$ and
$Eval(Expr, \delta' + VAR \Rightarrow Item) = e$. Then by I.H. on PathExpr, know that $Item \in CE(l)$ and C-AnnMap[l] = true, therefore our theorem holds. $\square$

We can similarly show that Theorem D.1(c) also holds.

D.2 Equivalence of QPT

Given a query expression E that conforms to our grammar, if $UEXPR$ is the universe of such expressions, and E_{core} is the normalized expression of E using the core grammar, then we show the following the theorem.

THEOREM D.6 (EQUIVALENCE OF QPT). $\forall E \in UEXPR$, *GenerateQPT(E)* $=$ $GenerateQPT(E_{core})$

PROOF. There are five cases to consider depending on types of the expression E.

Base case: E = (fn:doc(Name)| VAR|.)

In this case $E = E_{core}$, and therefore the theorem is vacously true.

Indunction Hypothesis: Suppose the lemma holds for sub-expressions of E. We now prove the lemma also holds for E.

Case 1: E = (fn:doc(Name)| VAR|.) '/' PathTailExpr

If $E' = (fn: doc(Name)|VAR|.)$, then $E_{core} = for\ \$dot\ in\ E'\ return\ PathTailExpr$. (Note the variable $dot and '.' in our grammar indicate the same context item).

First by Lemma D.2, we know that $|GenerateQPT(E')| = 1$. Also, by the argument of Case B.2 in Theorem D.1, we know that $|GenerateQPT(E_{core})|$. Assume GenerateQPT(E) = $\{Q_e\}$, GenerateQPT(E_{core}) = $\{Q_c\}$, and GenerateQPT(E') = $\{q'\}$.

By Lemma D.2, we know that |GenerateQPT (PathTailExpr) | = 1. Assume $\{q\} = GenerateQPT(\ PathTailExpr)$, and By line 7 in Figure 21, we know that $q'.V = \{q'.root\}$ and $q'.E = \emptyset$. If $E'' = \{(q'.root, l, '/', ann)|l \in q.root.Children\}$ and $E''' = \{(q.root, l, '/', ann)|l \in q.root.Children\}$ and, then by lines 11-16 in Figure 21, we know that $Q_e.V = (q.V - \{q.root\}) \cup \{q'.root\}$, $Q_e.E = (q.E - E''') \cup E''$ and $Q_e.root = q'.root$.

On the other hand, by line 7 in Figure 21, it is easy to see that $q.root = '\ .'$. Therefore in GenerateQPT(E_{core}), by lines 19-26 in Figure 24, the algorithm will create edges from $q'.root$ to the child nodes of $q.root$. Therefore $Q_c.V = (q.V - \{q.root\}) \cup \{q'.root\}$, $Q_c.E = (q.E - E''') \cup E''$ and $Q_c.root = q'.root$. Therefore $Q_E = Q_C$.

Case 2: E = PathExpr '[' PredExpr ']'

In this case, E_{core} = *for \$dot in PathExpr return if PredExpr then \$dot else ()*

There are two cases according to whether $dot is referenced in PredExpr. First, by line 40 in Figure 21, we know that GenerateQPT *(if PredExpr then \$dot else ())* $= GenerateQPT(PredExpr)\ \cup GenerateQPT('.')$.

First, if $\forall q \in GenerateQPT(PredExpr), q.root \neq '\ .'$. Then by lines 19-26 in Figure 24, we know that $Q_C = GenerateQPT(PathExpr) \cup GenerateQPT(PredExpr)$. GenerateQPT('.') is not in Q_C because it only has a single root node and therefore is ignored (lines 19-26). On the other hand, we know that if $r_q \neq '\ .'$, then line 24-25 in Figure 21 will not be executed, and therefore Q_E
$= GenerateQPT(PathExpr)\ \cup GenerateQPT(PredExpr)$. Consequently $Q_E = Q_C$.

Second, if $\exists q \in GenerateQPT(PredExpr)\ q.root = '\ .'$. Let $X = \{x \in GenerateQPT(PredExpr)\ x.root = '\ .'\}$, $\forall x \in X$, if $\{q\} = GenerateQPT(PathExpr)$, $E'' = \{(x.root, l, '/', ann)$ $|l \in x.root.Children\}$ and $E''' = \{(q.root, l, '/', ann)$ $|(x.root, l, '/', ann) \in E''\}$, then by lines 19-26 in Figure 24, we know that $Q_C = \{Q'\} \cup \{y|y \in GenerateQPT(PredExpr) - X\}$ where $Q' = (V', E', r')$ and $V' = \cup\{x.V - \{x.root\}|x \in X\}$, $E' = (q.E - E''') \cup E''$ and $r' = q.root$.

On the other hand, note that when invoking GenerateQPT(E), in Figure 24, lines 24-25 are essentially identical to lines 19-26, hence if $\exists q \in GenerateQPT(PredExpr)$ $q.root = '\ .'$, $Q_e = \{Q'\} \cup \{y|y \in GenerateQPT(PredExpr) - X\}$, and consequently $Q_E = Q_C$.

Case 3: for VAR in PathExpr where Expr1 return Expr2

In this case, E_{core} = for VAR in PathExpr return if Expr1 then Expr2 else ()

First, by line 40 in Figure 21, we know that GenerateQPT(if Expr1 then Expr2 else ())
= GenerateQPT(Expr1) $\cup$ GenerateQPT(Expr2). Let $G = GenerateQPT(Expr1)\ \cup GenerateQPT(Expr2)$. Then there are two cases according to whether VAR is referenced in G.

First, if $\forall g \in G\ g.root \neq VAR$. Then by lines 19-26 in Figure 21, we know that Q_C
$= GenerateQPT(PathExpr) \cup G$. On the other hand, we know that if $g.root \neq '\ .'$, lines 19-26 in Figure 21 will not be executed, and therefore $Q_E = GenerateQPT(PathExpr) \cup G$. Consequently $Q_E = Q_C$.

Second, if $\exists g \in G\ g.root = '\ .'$. Let $X = \{x \in G\ x.root = '\ .'\}$, $\forall x \in X$, if $\{q\} = GenerateQPT(PathExpr)$, $E'' = \{(x.root, l, '/', ann)|l \in x.root.Children\}$ and
$E''' = \{(q.root, l, '/', ann)|l \in x.root.Children\}$, then by lines 19-26 in Figure 24, we know that $Q_C = \{Q'\} \cup \{y|y \in G - X\}$ where $Q' = (V', E', r')$ and $V' = \{x.V - \{x.root\}|x \in X\}$, $E' = (q.E - E''') \cup E''$ and $r' = q.root$.

On the other hand, note that when invoking GenerateQPT(E), by lines 6-12 in Figure 24, we also first produce a set of QPT $G' = GenerateQPT(Expr1) \cup GenerateQPT(Expr2)$, therefore using the same argument on G' as above and using the same notations (with G' in place of G), we can infer that $Q_E = \{Q'\} \cup \{y | y \in G' - X\}$ where $Q = (V', E', r')$ and $V' = \{x.V - \{r_x\} | x \in X\}$, $E' = (q.E - E''') \cup E''$ and $r' = q.root$. and consequently $Q_E = Q_C$.

Case 4: (forClause|letClause)+ return Expr)

This is proved separately in Theorem D.7.

Case 5: Other cases

In all of other cases $E = E_{core}$ and therefore the theorem is vacuously true.

□

THEOREM D.7 (EQUIVALENCE OF QPT OF FLOWR). *For all E = (forClause|letClause)+ return Expr, GenerateQPT(E) = GenerateQPT(E_core)*

PROOF. For notationl convenience, let $Q_E = GenerateQPT(E)$ and $Q_C = GenerateQPT(E_{core})$.

We prove the lemma by inductions on the number of for/let clauses, denoted by d.

Base case: d=1 In this case, E = for VAR in PathExpr return Expr or E = let VAR := PathExpr return Expr.

In both cases $E = E_{core}$ and therefore the lemma is vacuously true.

Induction hypothesis: Assume the lemma holds for $d \leq n$. We now show the lemma holds for $d = n + 1$.

There are two cases, one for each different types of root clauses.

Case 1: E = for VAR in PathExpr (forLetClause)+ return Expr In this case, if E'=(forLetClause)+ return Expr, then E_{core} = for VAR in PathExpr return E'_{core}

By I.H., we know that GenerateQPT(E') = GenerateQPT(E'_{core}) = G. Note that we use the same lines of code (lines 19-26 in Figure 24) to handle G in GenerateQPT(E) and GenerateQPT(E'), therefore it is easy to see that $Q_E = Q_C$.

Case 2: E = let VAR := PathExpr (forLetClause)+ return Expr The proof of this case is identical to Case 1 due to line 33 in Figure 24.

□

E. GENERALIZED GENERATEPDT ALGORITHM

In this section we show the generalized algorithm GeneratePDT that handles the optimizations and extensions mentioned in Section 4.2.2.1.

Description of the algorithm

Figure 25 shows the high-level pseudo-code of our algorithm which addresses challenges described above. The algorithm takes in an 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 5-7).

At a high level, the Candidate Tree, or CT, is a tree data structure which consists of candidate nodes for the result PDT. CT nodes are created in document order for every

```
1:  GeneratePDT (QPT qpt, PathIndex pindex, KeywordSet
    kwds, InvertedIndex iindex): PDT
2:    pdt ← ∅
3:    (pathLists, invLists) ← PrepareLists(qpt, pindex, iindex,
      kwds)
4:    {Initialize CT}
5:    for idlist ∈ 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 ← CT.LeftMostPath
11:     for all cqn ∈ lmp do
12:       for all qn in cqn.CTQNodes where ∃l ∈ 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
```

Figure 25: Algorithm for generating PDTs

```
1:  AddCTNode(CTNode parent, DeweyID id, int depth)
2:    if depth ≤ id.Depth then
3:      curId ← Prefix(id, depth); qNodes ← QNodes(curId)
4:      if qNodes = ∅ then AddCTNode(parent,id,depth+1)
5:      else
6:        newNode ← parent.findChild(curId)
7:        if newNode = null then
8:          newNode ← 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 ∀i, q.DM[i]=1 then
17:       set DM[q] to 1 for nodes in q.PL
18:     end if
19:   end for
```

Figure 26: Algorithm for adding new CT nodes

Dewey ID in the lists and stores sufficient information for efficiently checking ancestor and descendant restrictions. Every CT node is a 3-tuple (ID, PdtCache, CTQNodeSet). For a CT node cn, cn.ID is the unique identifier of the node and always corresponds to a component of a Dewey ID in **pathLists**; cn.PdtCache stores the descendant nodes of cn that satisfy the descendant restrictions and do not satisfy the ancestor restrictions *yet*. Such nodes are stored in cn.PdtCache because the corresponding CT nodes are removed from the CT, and they have a chance to satisfy all restrictions when more Dewey IDs are processed (and hence they must be cached for further consideration).

Further, cn.CTQNodeSet stores the set of QPT nodes that cn.ID corresponds to. For example, in Figure 8, the id 1.2.1 corresponds to the QPT node with the tag name "isbn", and hence this node is in CTQNodeSet of the CT node with the id 1.2.1. In most cases cn.CTQNodeSet is a singleton set as in the above example; however, if the QPT path contains "//" axes (e.g., //a//a), or two different QPT paths have the same prefixes (e.g., a/b/c and a/b/d), cn.CTQNodeSet may contain more than one node because one single Dewey ID matches 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. 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.

More specifically, each item in CTQNodeSet is itself a 4-tuple (QNode, InPdt, ParentList (or PL), DescendantMap (or DM)). For an item q in cn.CTQNodeSet, q.QNode is one of the QPT nodes to which cn corresponds (as described earlier); q.InPdt is a boolean flag indicating whether the containing CT node cn $\in$ PE(q, D); q.PL stores items in CTQNodeSet of cn's ancestor nodes where the QPT nodes of these items are the parent node of q in the QPT. More formally, q.PL = $\{$*aq* $\in$ *acn.CTQNodeSet* $|$ *acn* $\in$ *ancestors(cn)* $\wedge$ *((aq.QNode, q.QNode, '/', ann)* $\in$ *Q* $\vee$ *(aq.QNode, q.QNode, '//', ann)* $\in$ *Q)* $\}$. Last, q.DM:$QNode \rightarrow bit$ keeps track of whether a node satisfies descendant restrictions. Intuitively, the value of each entry indicates if the item in CTQNodeSet has the mandatory child/descendant nodes that are the candidate elements. More formally, given a CT node cn, q $\in$ cn.CTQNodeSet, (q, cq, '/', 'm') $\in$ QPT $\Rightarrow$ (q.DM [cq] = 1 $\Leftrightarrow$ $\exists ch$ $\in$ CT, parent(n, ch) $\wedge$ ch $\in$ CE(cq, D)), and (q, cq, '//', 'm') $\in$ QPT $\Rightarrow$ (q.DM[cq] = 1 $\Leftrightarrow$ $\exists ch$ $\in$ CT, anc(n, ch) $\wedge$ ch $\in$ CE(cq, D)). Hence an item satisfies the descendant restrictions when its DM has the value 1 in all entries. Figure 28(b) illustrates the structure of a CT node that will be used later when we walk over the algorithm.

Now we go back to our algorithms. The algorithm invokes the routine AddNewCTNodes() (Figure 26) to create CT nodes for a given Dewey ID when necessary. After all of the corresponding CT nodes are created, the values of their DM's are updated from bottom up(line 17). Intuitively, when child/descendant nodes turn into candidate elements (either because they correspond to leaf node in the QPT, or all entries in DM have the value 1), they notify their parent/ancestor nodes the existence of themselves. Such information will be used to check the descendant restrictions of the parent/ancestor nodes.

After initializing the CT, the algorithm makes a single loop over the ID lists (lines 8-19 in Figure 27). Specifically, the algorithm performs three tasks at each step of the loop. First, it retrieves next minimum ids corresponding to the left most path of the CT and creates CT nodes corresponding to these ids. Second, it processes the CT nodes in the left most path from top down. Specifically, given a CT node n in the left most path and given an item q $\in$ 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 = $\emptyset$ or $\exists p \in$ 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 (line 2-5). However, if q satisfies the descendant restrictions but not the ancestor restrictions, then we temporarily create a corresponding node in the pdt cache of its parent CT node (lines 7-11). As described earlier, intuitively we defer the decision on the current node as late as possible to after we process all the relevant Dewey IDs (in fact, only when we remove the parent node will we process this cache node).

After processing all nodes in the left most path from top down, the algorithm starts removing nodes from the left most path from bottom up if the node does not have child nodes. The intuition behind this removing is that if the node satisfies the PDT definition, it is already output to the PDT in the second phase; if it does not satisfy the descendant restrictions, then since all the possible descendant IDs have been taken into account (because we retrieve IDs in document order), therefore it is safe to conclude that the node will not be in the PDT without looking ahead for more (irrelevant) Dewey IDs; if it satisfies the descendant restrictions but not the ancestor restrictions, then in the second phase, we already cache it in the pdt cache of its parent node and hence we can still remove it safely. Further, when we remove a node, we also check the nodes in its pdt cache. First, remember that the cache node already satisfies the descendant restrictions. Hence if we determine that it now also satisfies the ancestor restrictions, we immediately create the corresponding node in the result PDT (line 21). If the cache node still does not satisfy the ancestor restrictions, in general we propagate it to the pdt cache of the parent again, and therefore indicating more Dewey IDs are required to make a decision. However, if the cache node only depends on the node to be removed which itself does not (and will not, as explained earlier) satisfy the descendant restrictions, it is safe to determine that the cache node will not be in the result PDT, and therefore it can be removed (line 26).

To summarize, at each step, the algorithm ensures the following invariant: the Dewey IDs that are known to be PDT nodes are either in the candidate tree or in the result PDT (hence we do not miss any potential PDT nodes); and the result PDT only contains nodes that satisfy the PDT specifications. Finally, the algorithm terminates after exhausting

```
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
```

Figure 27: Algorithm for generating PDTs

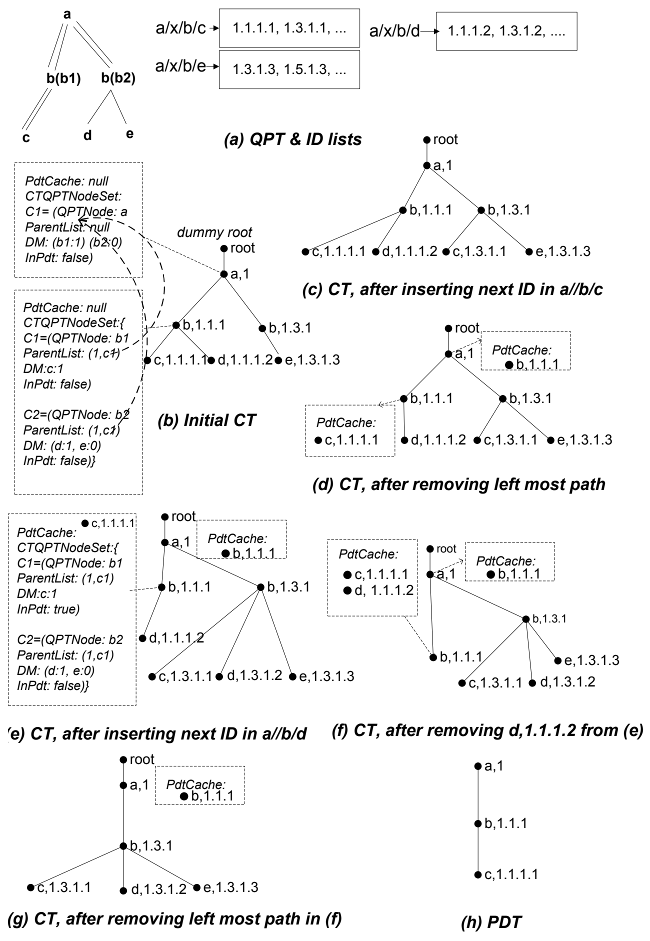


Figure 28: Generating PDT

all the IDs in the lists and exhausting all of the CT nodes. Hence by the invariant, the result PDT contains all and only the IDs that satisfy the PDT specifications.

Illustration of the algorithm

Now we walk through the algorithm using the QPT and ID lists shown in Figure 28(a). We use this example because it illustrates more features of our algorithm than our running example. Also for ease of exposition, we do not consider data values and tf values. Keep in mind they will be propagated along with Dewey IDs. 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 5-7). In **AddNewCTNode()**, CT nodes are created for distinct Dewey ID components that matches 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 can still be easily determined using their respective Dewey IDs.

Further, as mentioned earlier, new nodes will update DM of nodes in their PL. Figure 28(b) 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 Figure 28(a)), contains two items $c1$ and $c2$ in its CTQNodeSet. c1.DescedantMap[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 8-19). As mentioned earlier, it first retrieves next IDs corresponding to the left most path "a//b/c". Figure 28(c) shows the content of the CT at this point. The algorithm then inspects CT nodes from top down on the current left most 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 left most 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 28(d) 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 28(e) 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 28(g) 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 enforce. Figure 28(f) 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.

F. CORRECTNESS OF GENERATEPDT

Now we show that given a QPT, the algorithm GeneratePDT generates the correct PDT that conforms to our PDT definition. Theorem F.1 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 enviroment $\delta \in UE(D, Q)$, 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 correspend 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 F.1 shows the correctness of the algorithm GeneratePDT.

THEOREM F.1. *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, GeneratePDT(Q, δ(Q.root).PathIndex, δ(Q.root).InvIndex, KW) = PDT(Q, KW, δ).*

F.1 Notations

We now introduce more notations before proving the Theorem F.1.

Prefixes

Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, $\delta(Q.root) = d$, *(strLists, invLists) =PrepareList(Q, d.PathIndex, d.InvIndex, KW).* At a given time # t, we say $H(t, strLists) = \{id \in l | l \in strLists \wedge id \text{ is retrieved by the time } \# \ t\}$ is the set of ids that has been retreived from $strLists$ by the time # t (including t). In our algorithm, t corresponds to the number of loops (lines 8-19 in Figure 25).

Next, given a QPT node q in Q, for all q' in ancesctor nodes of q, and an dewey id did in $strLists$ corresponding to q, we use $Prefix(q, did, q')$ to denote the set of prefixies 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 $\in$ Prefix(l.QNode, lid, q) | lid $\in$ l, q $\in$ anc(l.QNode)} is the set of prefixes of ids in l w.r.t l.QNode, and Prefix(L) = {x $\in$ Prefix(l) | l $\in$ L}. is the set of prefixes of ids in H(t, strLists).

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) $\Leftrightarrow$ (m.id, n.id $\in$ Prefix(L) $\wedge$ 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), $\exists$ n $\in$ T, id(n) $\in$ T.Cont $\wedge$ 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), $id(n) = pid \Rightarrow$ Value(pid) = null $\Rightarrow$ Value(Node(T, pid)) = null.

Hence T(H(t, strLists)) denotes the hyperthetical of 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 begining of the loop # t by lines 10-14. 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 $\in UL$, we say the predicate Qualified(pid, Q, KW, L) = true $\Leftrightarrow$ pid $\in$ PDT(Q, KW, {Q.root $\Rightarrow$ T(L).root}).

F.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 mandantory 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 8-19 in Figure 25) 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 F.2. *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, 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 F.2 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)

F.2.1 Supporting lemmas for Lemma F.2

We now present a set of lemmas that will be used in the proof of Lemma F.2. Proofs will be presented after we show the main theorem.

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

LEMMA F.3 (MONOTONICITY). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 mandantory 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 mandantory 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 F.4 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 F.4 (DESCENDANTMAP). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 8-19), we start by adding new IDs corresponding to the current left most path, Then it is easy to infer the following lemma from Lemma F.4.

LEMMA F.5 (DM). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 F.6 indicates that if the flag InPdt of a CT node is true, the the id of this node is qualified.

LEMMA F.6 (INPDT). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 F.7 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 Figure 27 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 F.7 (PDTCACHE). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 F.8 (COMPLETENESS OF CT). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, if $\delta(Q.root) = d$ and (strLists, invLists)*

= PrepareList(Q, d.PathIndex, d.InvIndex, KW), then at the loop # t, ∀pid ∈ Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists))=false ∧∃L ∈ Comp(H(t, strLists)), Qualified(pid, Q, KW, L)=true ⇒ pid ∈ CT(t) ∨ pid ∈ pdtCache(CT(t)).

Lemma F.8 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 5-6 in Figure 25), 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 F.9 (Initialization of CT). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after initializing the candidate tree CT,*

(a) ∀id ∈ CT, ∃l ∈ strLists, ∃q ∈ anc(l.QNode), ∃pid ∈ Prefix(l.QNode, MinimumID(l), q), id = pid

(b) ∀l ∈ strLists, ∀q ∈ anc(l.QNode), ∀pid ∈ Prefix(l.QNode, MinimumID(l), q), pid ∈ CT.

F.2.2 Proofs of Lemma F.2

We separate Lemma F.2 into two parts and prove each of them separately.

Lemma F.10. *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after the loop # t, ∀pid ∈ Prefix(H(t, strLists)), Qualified(pid, Q, KW, H(t, strLists))=true ⇒ (pid ∈ GenPDT(t) ∨ pid ∈ CT(t)) ∨∃n ∈ CT(t).V, pid ∈ n.PDTCache (qualified nodes are in the candidate tree or the PDT).*

Lemma F.11. *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and (strLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW), then after the loop # t, ∀id ∈ GenPDT(t), id ∈ Prefix(H(t, strLists)) ∧ Qualified(id, Q, KW, H(t, strLists)) = true. (all nodes in the PDT are qualified).*

We first prove Lemma F.10.

Proof. We prove Lemma F.10 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 ∀*n* ∈ *CT*(*t*), n.PDTCache = null. On the other hand, by Lemma F.9, we know that ∀*l* ∈ strLists, ∀*q* ∈ anc(l.QNode), ∀*pid* ∈ Prefix(l.QNode, MinimumID(l), q), *pid* ∈ *CT*. This implies that ∀*pid* ∈ Prefix(H(0, strLists)), pid ∈ CT(0) and hence Lemma F.10 is vacously 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* ∈ *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* ∉ $CT((n+1)--)$.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* ∈ $CT((n+1)--)$.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 F.4, 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 F.6, 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 F.8, 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 F.11.

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

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

Indunction 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 F.3, we know Qualified(id, Q, KW, H(n+1, strLists)) = true, and hence the lemma holds.

Now we prove the lemma for all $g \in$ (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 $\exists q \in$ g.CTQNodes, q.InPdt = true, and hence by Lemma F.6, Qualified(g.id, Q, KW, H(n+1, strLists)) = true.

Case 2: $\exists cn \in CT((n+1)--)$*, g.id* $\in$ *cn.PDTCache.* Since g is created in GenPDT(n+1), by line 21 in Figure 27, we know that either (1) PL(g) = $\emptyset$ or (2) $\exists p \in$ PL(g), p.inPDT = true. Hence by Lemma F.7, we know that Qualified(g.id, Q, KW, H(n+1, strLists)) = true.

Case 3: g.id $\in$ *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.

□

F.2.3 Proofs of supporting lemmas for Lemma F.2

Proof of Lemma F.4

Proof. First, if MC(cnq) = $\emptyset$, then the lemma is vacously true and hence we only consider the case where MC(cnq) $\neq \emptyset$.

"$\Rightarrow$"

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$\leq$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 8-19. 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') $\in$ Q.E $\Rightarrow \exists$l $\in List_n$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_n$).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 $\in List_n$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_n$).root), ce.id = cid $\wedge$ id(cn) $\in$ Prefix(l.QNode, lid, cnq.QNode) $\wedge$ anc(id(cn), cid)).

Hence by the definition of candidate elements, it is easy to infer that ((cnq.QNode, qcd, '/', 'm') $\in$ Q.E $\Rightarrow \exists$l $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_{n+1}$).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 $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_{n+1}$).root), ce.id = cid $\wedge$ id(cn) $\in$ Prefix(l.QNode, lid, cnq.QNode) $\wedge$ 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$\in$ l the id we add at the time n+1, then we can infer that $\exists qid \in$ 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 $\exists qid \in$ CE(qcd, T($List_{n+1}$).root). Further since we set cnq.DM[qcd] = 1, we know that cqn $\in$ qcd.PL, therefore we can finally conclude that (cnq.QNode, qcd, '/', 'm') $\in$ Q.E $\Rightarrow \exists$l $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_{n+1}$).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 $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, qcd), $\exists ce \in$ CE(qcd, T($List_{n+1}$).root), ce.id = cid $\wedge$ id(cn) $\in$ Prefix(l.QNode, lid, cnq.QNode) $\wedge$ anc(id(cn), cid)).

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

If MC(qcd) = $\emptyset$, then we can use the similar argument as the base case and show the lemma holds. Otherwise MC(qcd) $\neq \emptyset$.

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 $\exists qn \in CT_n$, $\exists qc \in$ cn.CTQNodeSet, qcd = qc.QNode $\wedge$ $\forall mq \in$ 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$\in$MC(qcd) *wedge* qc.DM[x] = 1 in CT_n}, and Y = {y|x$\in$MC(qcd) $\wedge$ 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, $\forall mq \in$ MC(qc), (qcd, mq, '/', 'm') $\in$ Q.E $\Rightarrow \exists$l $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, mq), $\exists ce \in$ CE(mq, T($List_{n+1}$).root), ce.id = cid $\wedge$ id(qn) $\in$ Prefix(l.QNode, lid, qcd) $\wedge$ parent(id(qn), cid)) $\wedge$
((cnq.QNode, qcd, '//', 'm') $\in$ Q.E $\Rightarrow \exists$l $\in List_{n+1}$, $\exists$lid $\in$ l, $\exists$cid $\in$ Prefix(l.QNode, lid, mq), $\exists ce \in$ CE(mq, T($List_{n+1}$).root), ce.id = cid $\wedge$ id(cn) $\in$ Prefix(l.QNode, lid, qcd) $\wedge$ anc(id(cn), cid)).

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

Second, if $\nexists qn \in CT_n$, $\exists qc \in$ 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).

"$\Leftarrow$"

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 seperately.

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) ≠ ∅, and ∃*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 ∀*x* ∈ *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* ∈ *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. ∃*qn* ∈ cnn.CTQNodeSet, qn.QNode = qcd, then ∀*x* ∈ MC(qcd), qn.DM[x] = 1. And hence by the algorithm, we will set cnq.DM[qcd] to be 1.

□

Proof of Lemma F.6

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

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

Indunction 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 ∀*cnq* ∈ cn.CTQNodeSet, 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 Figure 27, cnq.inPdt is set to true when ∀*i* ∈ cnq.DM[i] = 1. By Lemma F.4, 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.

Indunction 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 Figure 27, we know that ∀*i* ∈ cnq.DM[i] = 1. By Lemma F.4, this implies that cn ∈ CE(cnq.QNode, T(H(t+1, strLists)).root). We also know that cnq.PL = ∅ or ∃*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 ∃*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 F.3 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 ∀*n* ∈ *CT*(*t*−).LeftMostPath, ∀*nq* ∈ n.CTQNodeSet, cn ∈ PE(nq.QNode, T(H(t+1, strLists)).root)) ⇒ nq.InPdt = true.

It is straightforward to infer that ∀*n* ∈ *CT*(*t*−).LeftMostPath, ∀*nq* ∈ n.CTQNodeSet, cn ∈ PE(nq.QNode, T(H(t, strLists)).root)) ⇒ nq.InPdt = true ∧ (∀*t'* ≥ *t*, n ∈ CT(t') ⇒ nq.InPdt = true ∧ n ∈ *CT*(*t'*−) ⇒ (nq ∈ n.CTQNodeSet ∧ nq.InPdt = true) ∧ n ∈ *CT*(*t'*−−) ⇒ (nq ∈ n.CTQNodeSet ∧ 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 ∀*cnq* ∈ 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 ∈ PE(sq.QNode, T(H(t, strLists)).root), we can infer that cn ∈ CE(sq.QNode, T(H(t, strLists)).root). Hence by Lemma F.4, ∀*i* ∈ cnq.DM[i] = 1. Further, since sq is the root node in the QPT, sq.PL = ∅. Then by line 4 in Figure 27, cnq.inPdt is set to true. Hence (b) holds.

Indunction 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 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* ∈ cnp.CTQNodes, then *cn* ∈

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 F.4, $\forall i \in$ cnq.DM[i] = 1. Therefore by lines 4-5 in Figure 27, 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 F.7

Proof. (a) It is easy to prove (a) by lines 11 in Figure 27 using Lemma F.4.

(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 Figure 27. 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 F.8

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 F.3, 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 $\in$ 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 Figure 27. 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 14 in Figure 27, we have already added next minumum ids corresponding to the left most path. Also, for all paths in the lists, the next minumum 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 $\in$ 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 F.4, we know that $\forall L \in$ Comp(H(n+1, strLists)), pid $\notin$ CE(pn.CTQNode, T(L).root)), and hence Qualified(pid, Q, KW, L) = false.

Second, if pid $\in$ 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 $\in$ 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 Figure 27. Therefore we know that before we remove pn, pn.PL =\{cn\} and $\exists ch \in$ MC(cn), cn.DM[ch] = 0. By Lemma F.4 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 $\in$ 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 14 in Figure 27) 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 $\in$ 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.

□

F.2.4 Proofs of correctness of **PrepareList**

By Lemma F.2, 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 $\Rightarrow$ T(strLists).root\}). We just need the following final lemma to show the Theorem F.1

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 F.12 (PATHINDEX). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 F.13 (CANDIDATEELEMENTS). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 ⇒ d}) ⇒ $n \in$ 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, $\forall n \in$PE(q, {Q.root ⇒ d}). Since q does not have children nodes, we will issue d.PathIndexLookUp(RootToLeaf(q, Q)). Therefore by Lemma F.12, 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 ⇒ d}), $\forall cq \in$ MC(q), (q, cq, '/', m) ∈ Q ⇒ $\exists nc \in$ PE(cq, {Q.root ⇒ d}), parent(n, nc) ∧ (q, cq, '//', m) ∈ Q ⇒ $\exists nc \in$ PE(cq, {Q.root ⇒ d}), anc(n, nc).

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

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

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

Hence the lemma holds □

LEMMA F.14 (PREPARELIST). *Given a set of keyword KW, a QPT Q, an XML database D, an enviroment $\delta \in UE(D,Q)$, 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 ⇒ d}) ⇔ $n \in$ PE(q, {Q.root ⇒ T(strLists).root}).*

PROOF. "⇒"
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 ⇒ d}) = CE(q, {Q.root ⇒ d}), and PE(q, {Q.root ⇒ T(strLists).root}) = CE(q, {Q.root ⇒ T(strLists).root}). By Lemma F.13, we know PE(q, {Q.root ⇒ d}) ⊆ CE(q, {Q.root ⇒ T(strLists).root}), and hence PE(q, {Q.root ⇒ d}) ⊆ PE(q, {Q.root ⇒ T(strLists).root}). Therefore the lemma holds in the base case.

Induction Hypothesis: Assume the lemma holds for q of depth ≤ 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) ∈ Q, the case where (p, q, '//', ann) ∈ Q can be shown similarly.

First, by Lemma F.13, $\forall n \in$ PE(q, {Q.root ⇒ d}), $n \in$ CE(q, {Q.root ⇒ T(strLists).root}). Then by definition, we know that $\forall n \in$ PE(q, {Q.root ⇒ d}), $\exists np \in$ PE(p, PE(q, {Q.root ⇒ d}), parent(np, n). Hence by I.H. on nq, we know that $\forall n \in$ PE(q, {Q.root ⇒ d}), $\exists np \in$ PE(p, {Q.root ⇒ T(strLists).root}) parent(np, n). Hence $\forall n \in$ PE(q, {Q.root ⇒ d}), $n \in$ CE(q, {Q.root ⇒ T(strLists).root}). ∧ $\exists np \in$ PE(p, {Q.root ⇒ T(strLists).root}) parent(np, n).

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

Hence the lemma holds.

"⇐"
This direction follows from Lemma F.3 because $\forall id \in$ strLists, $id \in$ D. Hence the full proof is skipped.

□