The *Collection Index* to Support Complex Approximate Queries on XML Documents

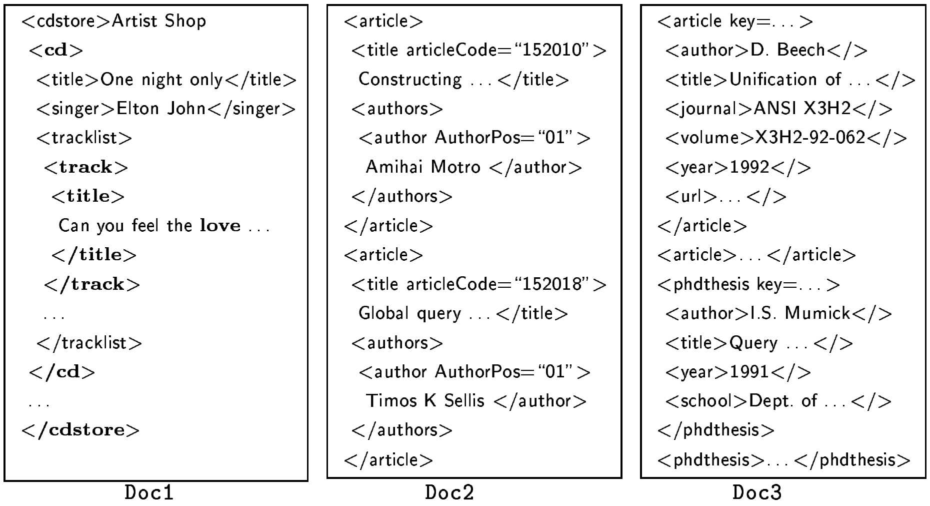
Paolo Ciaccia and Wilma Penzo

DEIS - IEIIT-BO/CNR
University of Bologna, Italy
{pciaccia,wpenzo}@deis.unibo.it

Abstract. The presence of structure in XML documents poses new challenges for the retrieval of data. Answering complex structured queries with predicates on *context* where data is to be retrieved, implies to find results that match *semantic* as well as *structural* query conditions. Then, the structural heterogeneity and irregularity of documents in large digital libraries make necessary to support *approximate queries*, i.e. queries where matching conditions are *relaxed* so as to retrieve results that possibly partially satisfy user's query conditions.
Exhaustive approaches based on sequential processing of documents are not adequate as to response time. In this paper we present an indexing method to execute efficiently approximate complex queries on XML documents. Approximations are both on content and document's structure. The proposed index provides a great deal of flexibility, supporting different query processing strategies, depending on the constraints the user might want to set to possible approximations on query results.

1 Introduction and Related Work

XML is announced to be the standard for future representation of data, thanks to the capability it offers to compose semi-structured documents that can be checked by automatic tools, as well as the great flexibility it provides for data modelling. The presence of nested tags inside XML documents leads to the necessity of managing *structured* information. In this scenario, traditional IR techniques need to be adapted, and possibly redesigned, to deal with the structural information coded in the tags. When querying XML data, the user's is allowed to express structural conditions, i.e. predicates that specify the *context* where data is to be retrieved. For instance, the user might want to retrieve: "Papers having title dealing with *XML*" (Query1). Of course, the user is not interested in retrieving whatsoever is containing the keyword "XML". This implies to find both a *structural match* for the context (title of papers) and a (traditional IR) semantic match for the content (the "XML" issue) *locally* to the matched context. Then, the structural heterogeneity and irregularity of documents in large digital libraries, as well as user's ignorance of documents structure, make necessary to support *approximate queries*, i.e. queries where matching conditions are *relaxed* so as to retrieve results that possibly *partially* satisfy user's query



```
<cdstore>Artist Shop
 <cd>
  <title>One night only</title>
  <singer>Elton John</singer>
  <tracklist>
   <track>
    <title>
     Can you feel the love ...
    </title>
   </track>
   ...
  </tracklist>
 </cd>
 ...
</cdstore>
```

Doc1

```
<article>
 <title articleCode="152010">
  Constructing ...</title>
 <authors>
  <author AuthorPos="01">
   Amihai Motro </author>
 </authors>
</article>
<article>
 <title articleCode="152018">
  Global query ...</title>
 <authors>
  <author AuthorPos="01">
   Timos K Sellis </author>
 </authors>
</article>
```

Doc2

```
<article key=...>
 <author>D. Beech</>
 <title>Unification of ...</>
 <journal>ANSI X3H2</>
 <volume>X3H2-92-062</>
 <year>1992</>
 <url>...</>
</article>
<article>...</article>
<phdthesis key=...>
 <author>I.S. Mumick</>
 <title>Query ...</>
 <year>1991</>
 <school>Dept. of ...</>
</phdthesis>
<phdthesis>...</phdthesis>
```

Doc3

Fig. 1. Sample XML documents

conditions. Powerful query tools should be able to efficiently answer structural queries, providing results that are ranked according to possible relaxations on the matching conditions. As an example, consider document `Doc1` in Fig. 1 and the query: "Retrieve CDs with songs having the word *love* in the title" (Query2). *Doc1* contains a relevant answer that should be returned, although some dissimilarities are present with respect to the query structural condition: 1) The `track` element indeed can be considered a synonym for `song` in this context (*semantic relaxation*), and 2) an additional `tracklist` element is present between `cd` and `track` (*structural relaxation*). An exact match of the query on `Doc1` would have returned no result. The above query is more properly a *path query*, i.e. a query where structural condition is expressed by a single sequence of nested tags. Things are more complicated when answering complex queries, i.e. queries with two or more *branching* conditions. Finding structural approximate answers to this kind of query is known to be a hard task, namely, finding a solution to the *unordered tree embedding problem*[1] [5–7, 14, 15], which is proved to be NP-complete [10]. In [15] some critical assumptions of the problem are simplified to reduce the overall complexity, as to support approximate structural queries.

In this paper we present an indexing structure, the *Collection Index*, that effectively and efficiently supports *complex approximate queries* on XML data. The Index aims to reduce the complexity of finding approximate query patterns, avoiding the sequential examination of all documents in the collection. This issue has been recently covered by several approaches [8, 9, 11, 13]. However, all these works do not deal with semantic nor structural approximations, except for the explicit use of wildcards in query paths. Actually, wildcards do not contribute to result ranking: Data is asked to satisfy a relaxed pattern, no matter the "grade"

[1] According to the XML Information Set W3C Recommendation [4], XML documents can be represented as trees. Complex queries can be also represented as *pattern trees* to be searched in the documents.

of relaxation. Thus, for instance, *cohesion* of results is not taken into account for ranking. Approximation is investigated in [17], although the proposed method focuses only on *semantic* similarity. Structural relaxations are supported by the index structures proposed in [12], though these are limited to deal with path expressions. The indexing method we propose is based on an *intensional view* of the data, similarly in spirit with traditional database modeling where data is separated from the schema. The Collection Index is a concise representation of the structure of all the documents in the collection, in that each document can be mapped exactly in one part of the index. The structure we propose resembles a DataGuide [9], but overcomes its limitations.[2] The Collection Index efficiently supports queries on XML document collections producing relevant *ranked results* that even partially satisfy query requirements, also in absence of wildcards in the query. Results are ranked according to their approximation to both structural and semantic query conditions.

The paper is organized as follows: In Section 2 we compare our approach with ApproXQL [15], a similar system that supports complex approximate query processing on XML data. Section 3 presents the Collection Index, as an *extended template* for the structure of a set of XML documents. In Section 4 we show how the Collection Index *efficiently* and *flexibly* supports approximate query processing: Different query processing strategies can be applied on the Index, depending on the constraints the user might want to set to possible approximations on query results. Complexity of query processing is also discussed, showing the feasibility of our approach. In Section 5 we sketch some experiments on a large heterogeneous collection of XML documents [16], and finally in Section 6 we draw our conclusion and discuss future developments.

2 Comparison with ApproXQL

Supporting approximate complex queries are work in progress (or future work) of most of the mentioned proposals that deal with efficient techniques to query XML data [12]. To authors' knowledge, ApproXQL [15] is currently the more complete system that enables the user to retrieve all approximate results to complex structured queries. ApproXQL applies different approximations to the query tree, allowing for renaming, insertion, and deletion of query nodes. We will refer to this system to compare the efficiency and the effectiveness of our method. The focus of the problem is finding an efficient algorithm to solve the tree embedding problem [10], properly reformulated to escape NP-completeness.

Reformulating the Tree Embedding Problem. Both queries and documents are assumed to be tree-structured. We recall that, given two trees t_1 and t_2, an injective function f from nodes of t_1 to nodes of t_2 is an *embedding* of t_1 into t_2, if it preserves labels, and ancestorship in both directions [10].

In order to make the embedding function efficiently computable, ApproXQL, as well as our approach, guarantees ancestorship only in one sense, from query

[2] Basically, a DataGuide would have not been helpful in answering Query2, since it does not allow for approximate navigation.

tree to document tree. This means that siblingness is not guaranteed by the mapping, in that sibling nodes in the query may be mapped to ancestor-descendant nodes in the data. For convenience, we will use the term embedding to denote such reformulated tree embedding problem, according to the one-way ancestorship preservation. For the sake of simplicity, we limit to *insertion* of nodes the approximations on results. This is compliant with the definition of tree embedding (as to ancestorship). We will complete discussion on renaming and deletion of query nodes in Section 4.3.

According to this new formulation of tree embedding problem, the result of a complex query for the ApproXQL method is a set $\mathcal{R}$ of nodes of the document tree, such that each $r \in \mathcal{R}$ is the root of a document (sub)tree that matches the query root and contains at least one embedding of the query tree. Using the Collection Index, we return the set $\mathcal{S}$ of minimal[3] document (sub)trees that contain at least one embedding of the query tree. It is easy to show that the relationship between ApproXQL and Collection Index results is: $\mathcal{R} = \bigcup_i \text{root}(t_i)$ $t_i \in \mathcal{S}$. We can derive that both ApproXQL and the Collection Index retrieve (or provide a reference to) the same embeddings.

A View to Complexity. Processing a complex structured query[4] is proved to require time $O(n_q{*}l{*}s)$, with n_q number of query nodes, l maximal number of repetition of a given label along a document path, and s maximal number of occurrences of a label in the document tree [15]. The goal of the Collection Index is to reduce this complexity. We will show that our method obtains the same results as ApproXQL in time $O(n_q{*}l{*}s_c^I{*}p^I)$, with $s_c^I{*}p^I \leq s$, where s_c^I is the maximal number of *contextual occurrences* of a label l_i in all index subtrees rooted at a given label l_j (each subtree is called a *context* for label l_j), and p^I is the maximal number of occurrences of a given index path in the document tree. It is worth to note that in most cases $s_c^I{*}p^I << s$, depending on the heterogeneity of the document collection. More details are given in Section 4.2.

3 The Collection Index

The extensional representation of a large XML document collection presents a great deal of redundancy because of the repeated tags to specify data semantics and structural organization for each document. Consider Fig. 1, showing two sample XML excerpts from XML Sigmod [1] (`Doc2`) and dblp [2] (`Doc3`) collections.[5] Whilst redundancy is evident for homogeneous collections (`Doc2`), the presence of repeated structural information is very common also in heterogeneous collections (e.g. *article* and *phdthesis* elements in `Doc3`). In general, the collected documents can be partitioned into homogeneous groups, according to the same basic structure they agree upon. Basically, the Collection Index is an

[3] Among all document (sub)trees that contain n embeddings, the *minimal* subtree has no proper subtrees that contain the same n embeddings.

[4] Recall that here only insertions of data nodes are allowed.

[5] We consider a single large XML document gathering several records as a collection of basic documents, one for each XML record.

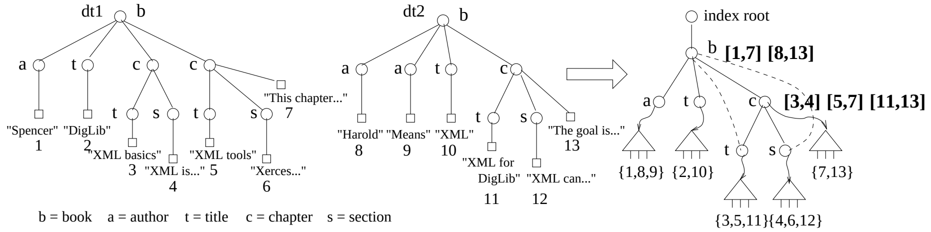


Fig. 2. Insertion of two documents in the index

extended template for the structure of a set of XML documents. The Index synthesizes the documents in the collection, and provides a skeleton in accordance with the structural relationships occurring in the documents.

3.1 Index Overview

The index is tree-structured: Nodes represent elements and attributes, and arcs define hierarchical relationships according to the nesting of elements/attributes in the documents. This structure forms the basic template of the document collection. We start simple, giving an intuition of index construction, and deferring a formal description of its components to Section 3.2.

The index is created incrementally, one document at a time. Before entering the documents in the index, a tree representation is provided for each document. Basically, each sequence[6] of tags $(t_1, \ldots, t_k)$, with t_1 document root, referencing a CDATA/PCDATA content defines a *path instance* in the document tree, i.e. a sequence of nodes $(n_1, \ldots, n_k)$ where n_1 is the tree root, such that $\forall i$, n_i is labelled with t_i, and n_k is a *tree leaf* that refers the data content. Given a path instance $p = (n_1, \ldots, n_k)$, the sequence $p^s = (label(n_1), \ldots, label(n_k))$ defines the *path schema* of p . Each referred CDATA/PCDATA section is stored in what we call a *data leaf.* All data leaves are *globally* numbered with increasing integer values from left to right, according to a depth-first visit of each document's tree. This numbering scheme allows for uniquely identifying each data leaf to be referenced by the index. Two sample document trees *dt1* and *dt2* are shown on the left side of Fig. 2: Squares denote data leaves, whereas tree nodes are marked by circles. The insertion of a document tree t_d in the index is equivalent to inserting (without repetitions) all the path instances that compose t_d. The path instances of t_d are assigned to *equivalence classes* according to the sequence of node labels (i.e. document tags) they are made of.

Definition 1 (Path Equivalence Class (PEC)). *A Path Equivalence Class Ξ is a set of path instances with the same path schema. Formally: Given two path instances $p_1 = (x_1, .., x_k)$ and $p_2 = (y_1, .., y_h)$, $\{p_1, p_2\} \subseteq \Xi \iff k = h$ and $\forall\ i \in [1..k]$ label(x_i)=label(y_i). For each path instance p_i $\exists$ a PEC Ξ_j such that $p_i \in \Xi_j$.*

Any (tree-structured) document collection $\mathcal{C}$ can be partitioned into a set of PECs, according to the path instances that compose documents in $\mathcal{C}$.

[6] Elements and attributes are considered equivalent. This implies that sequences might end with attributes.

Equivalence classes are created and grow as documents are processed. Equivalent path instances *share* the same path in the index, i.e. only one path per class is present in the index: This *template path* has the same schema of the path instances of the class, i.e. the *schema* of the class.

Definition 2 (PEC Schema). *Given any path instance $p \in \Xi$, the PEC schema Ξ^s of Ξ is p^s, the path schema of p.*

This leads to the following definition, which is the milestone of the structure of the Collection Index.

Definition 3 (Collection Index Path). *Given a document collection $\mathcal{C}$, a path p_I with schema p^s_I is a Collection Index Path for $\mathcal{C}$ $\Longleftrightarrow$ $\exists$ a PEC instance Ξ of $\mathcal{C}$, with schema Ξ^s, and $\Xi^s = p^s_I$. For each pair of PEC instances Ξ_i, Ξ_j, let p_i and p_j be their corresponding Collection Index Paths, and let p^s_M be the maximal prefix*[7] *of PEC schemas Ξ^s_i and Ξ^s_j. Then, p^s_M is also the maximal prefix of p^s_i and p^s_j and p_i and p_j share the (sub)path with schema p^s_M in the Collection Index.*

For instance, in Fig. 2 the two paths in *doc1* referring to data leaves numbered 3, and 5, and the path in *doc2* referring to data leaf numbered 11, respectively, belong to the same PEC Ξ_i having the same schema $\Xi^s_i = (b, c, t)$. On the other hand, the paths in *doc1* leading to data leaves numbered 4, and 6, and the path in *doc2* referring to data leaf numbered 12, respectively, belong to the same PEC Ξ_j, having the same schema $\Xi^s_j = (b, c, s)$. Ξ^s_i and Ξ^s_j share their maximal prefix (b, c) in the index.[8]

Each index (sub)path[9] references the set of data leaves referred by the document paths it has been derived from. In our example, the index path $/b/c/t$ references data leaves numbered 3, 5, and 11, and the index (sub)path $/b/c$ references data leaves numbered 7, and 13. The index provides an intensional view of the documents structure, where each occurrence of a path in a document is present only once in the index. The extensional representation of the document collection is given by the set of data leaves referenced by the index, since the documents structure is implicitly specified by the index itself. Each index (sub)path define a *semantic context* where the underneath data leaves are referenced according to IR access structures local to the (sub)path, e.g. B+-tree on leaf numbers, and inverted lists on terms. These access structures are depicted by triangles in Fig. 2. We call each structure a *value-index.*

[7] Given two sequences $s_1 = (x_1, .., x_k)$, and $s_2 = (y_1, .., y_h)$, the *maximal prefix* of s_1 and s_2 is the longest sequence s such that $prefix(s, s_1) \wedge prefix(s, s_2)$, where $prefix$ is defined as follows:

$$prefix(x, y) = \begin{cases} true \ \ \text{if } x = (x_1, \ldots, x_k) \\ \qquad \wedge\ y = (y_1, \ldots, y_k, y_{k+1}, \ldots, y_h) \\ \qquad \wedge\ x_i = y_i\ \forall i \in [1..k] \\ false\ otherwise \end{cases}$$

[8] The dashed lines and the numbers between square brackets keep additional information that will be explained in Section 3.2.

[9] We always refer to (sub)paths that start from root.

3.2 The Index Structure

In order to support data retrieval, the index skeleton sketched above needs additional information regarding the document instances indexed. After a document is added to the index, information on the new referenced data leaves is propagated to the inner nodes. Each inner node n keeps a pair of integer values $[l_{min}$,$l_{max}]$, for each occurrence of the (sub)path from index root to n occurring in the data tree. Each range denotes the set of leaves underneath the instance data node, such that l_{min} is the lowest leaf number, and l_{max} is the highest one, respectively. As an example, in Fig. 2 ranges are assigned to index nodes labelled "b" and "c", denoting books and chapters, respectively. The ranges assigned to index node "b" say that two occurrences of book elements are present in the collection, and that the data leaves they refer to are 1 to 7, and 8 to 13, respectively. Similarly, ranges assigned to index node "c" say that three occurrences of chapter elements are present in the collection, each one referring the sets of data leaves 3 to 4, 5 to 7, and 11 to 13, respectively.[10] Note that, each sequence of ranges is *locally ordered* by increasing values of l_{min}, and that ranges are *disjoint*. This is due to the numbering scheme of the data leaves when processing documents to generate the Collection Index paths.

Insertion of documents is an incremental process that does not require the index to be restructured. The index begins with a dummy root node. All the data paths are inserted starting from the root, which collects all the path entries. This allows for indexing also heterogeneous collections of documents. There is no need for a priori knowledge of the schema of the data, since the paths we encode are extracted from the data itself. Actually, the complete index is an *extended tree* in that it is more properly a single-rooted DAG. Each inner node, index root included, references its descendants in the tree. We take advantage of this information to efficiently perform *approximate* retrieval on structure.[11] In Fig. 2, arcs depicted by dashed lines denote ancestorship between nodes: For clarity, parentship is expressed by solid arcs, and references to descendants of the index root are omitted. Formally: Let $\mathcal{N}$ be a set of nodes, *NID* set of index node ids, *DID* set of (sub)document ids,[12] Λ set of node labels, Ψ set of document paths, and *VID* set of value-index ids. The Collection Index is defined as follows.

Definition 4 (Collection Index). A Collection Index is a tree $\Im = (\mathcal{N}, r)$, with $\mathcal{N} = \mathcal{N}_I \cup \mathcal{N}_L$, with $\mathcal{N}_I$ set of inner nodes, $\mathcal{N}_L$ set of leaves, and r $\in \mathcal{N}_I$ root of the tree, s.t.:

$$\mathcal{N}_I = \{n = (nid, l, \delta, \rho, vid) \mid n \in \mathit{NID} \times \Lambda \times 2^{\mathit{NID} \times \Lambda} \times 2^{\mathbb{N} \times \mathbb{N} \times \mathit{DID}} \times \mathit{VID}\}$$
$$\mathcal{N}_L = \{l = (lid, l, p, vid) \mid lid \in \mathit{NID} \times \Lambda \times \Psi \times \mathit{VID}\}$$

where $\forall\, n \in \mathcal{N}_I$, δ is a set of labelled id's denoting descendants of n in $\Im$, and ρ is a set of tuples $t = (min, max, did)$ each denoting a range $[min, max]$ originated

[10] Ranges can be omitted for index leaves since their values can be obtained from the value-indices.

[11] Of course, this choice is not costless. The overhead of this additional information is discussed in Section 3.2.

[12] Each node of a document tree is supposed to be uniquely identified.

from (sub)document *did* in the document collection; $\forall\, l \in \mathcal{N}_L$, $p \in paths(\Im)$ and $p = (r, \ldots, l)$.

Index Size As introduced in Section 2, we compare the Index Collection with ApproXQL [15]. In order to perform structural approximations, the Collection Index takes advantage of the lists of labelled descendant nodes δ assigned to each index node $n \in \mathcal{N}_I$. The list immediately points out nodes having a given label, located in specific parts of the index, namely in the subtree of the node n itself (denoting the scope of the search). The size of collecting these lists can be proved to be at most $O(n^I * h^I)$, with n^I number of nodes in the Collection Index, and h^I height of the Index. Depending on the grade of heterogeneity of the document collection, the value of n^I can be very small, in many cases comparable to the number of labels of a DTD for the collection. On the other hand, the value of h^I is generally small: it represents the maximal nesting level of tags in the documents. This means that the skeleton of the Index in most cases has a minimal impact as to its size. However, additional space is required for keeping information on ranges of data leaves assigned to each occurrence of document (sub)trees. The size of the ρ lists can be estimated as $O(n_d)$, with n_d number of nodes in the data tree. This evaluation is comparable with the size of the global inverted lists used in the ApproXQL approach, where each distinct label in the data is assigned the list of its occurrences in the document tree.[13] In comparison with the size of the ρ lists, in the general case, the quantity $O(n^I * h^I)$, stating the size of the Index skeleton, can be considered negligible.

4 Approximate Query Processing

We assume a query t_q to be tree-structured, expressing both semantic and complex structural conditions. Rather than attempting a tree embedding of t_q for each document, which is a time-consuming task, we rely on the Collection Index. Basically, the Index is navigated top-down following the arcs in accordance with the query structural conditions.

Various navigation strategies can be applied on the Collection Index, depending on the constraints the user might want to set to possible approximations on query results. We start presenting the navigation method to obtain approximate embeddings that relax the query conditions to possible insertion of nodes in the data. Other kinds of approximations, namely, renaming and deletion of query nodes are discussed in Section 4.3, together with the strategies to obtain the corresponding sets of approximate embeddings of the query tree.

4.1 Navigating the Collection Index

Approximate embeddings of the query tree in the Collection Index are retrieved in two steps, according to Algorithm `Solve` in Fig. 3 : 1) Top-down navigation for selecting relevant contexts, and 2) Bottom-up construction of final results.

[13] This computation includes also the inverted lists for the content-based retrieval of data leaves. As discussed in Section 3.1 we use inverted lists which are *local* to each structural context. However, their size is comparable to the size of the global lists used in [15] for content retrieval.

```
Solve(IndexTree I, QueryTree q)
begin
  q' ← SetMatches(I,q)
  FindEmbeddings(I,q')
end
```

Fig. 3. Algorithm `Solve`

```
SetMatches(IndexTree I, QueryTree q)
begin
1. for each $t_i$ in $\delta$ of I s.t. label($t_i$)=label(q)
2.    append (&I,&$t_i$) to list of context. matches of q
3.    for each $q_j \in$ children(q)
4.       $q_j$ ← SetMatches($t_i$,$q_j$)
5. return q
end
```

Fig. 4. Algorithm `SetMatches`

Exploration. The navigation step extends the query tree: Each query node is assigned a *list of matches* with nodes of the index tree. More precisely, each match is a *contextual match*, in that it is a pair $m = (\&context, \&node)$ such that $\&node$ is a pointer to the matching node in the index, and $\&context$ is a pointer to the index node that specifies the context where $\&node$ has been retrieved. Each context node is a match for the parent of the current query node. The context node of the query root is the Collection Index root.

The goal is to find all the *structural* matches of the query tree in the Collection Index. Semantic match of query leaves containing CDATA/PCDATA content is not considered at this step: This will be the starting point of Algorithm `FindEmbeddings` for the construction of results. Exploration of the Collection Index is guided by a query tree q as follows (algorithm `SetMatches`):

1. Given a context I in the Collection Index, descendant index nodes in I having the same label as the query root's label are set as matching nodes of q (lines 1 – 2). This means that only *relevant contexts* of the index are explored. Each match found is the root of a potential approximate embedding schema.[14]
2. The search proceeds recursively for each query node (lines 3 – 4), according to a pre-order visit of the query tree. Each matched index node found at the previous step is taken as the *current context* in the Collection Index.

Figure 5 presents the result of the `SetMatches` Algorithm to answer a structured query. For clarity, only node matchings are shown, depicted by dashed bold lines.

Construction of results Recall from Section 2 that the evaluation of a query q on a document d with the Collection Index returns a set of subtrees of d, each one denoting an approximate result for query q. Each returned subtree collects the set of approximate embeddings of q rooted at the same (sub)document *did* in d. These id's indicate the roots of the instances of the query tree in document d. Note that each (sub)document *did* that is the root of a returned subtree is in the list ρ of an index node that matched the query root. In order to determine which are the *did*'s in the ρ lists that actually represent the root of relevant results, semantic conditions of q are to be satisfied in the data leaves of the (sub)document instance. At this purpose, only *relevant* data leaves are considered, in that we take advantage of the value-indices of index nodes that matched

[14] Recall that the Collection Index provides an intensional representation of the structural organization of the data, basically it collects all structural schemas that may occur in the data.

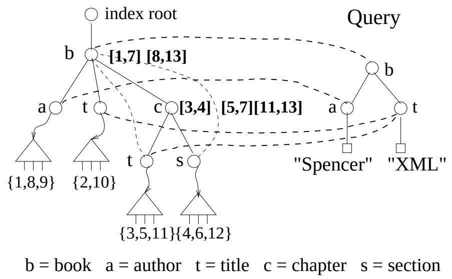
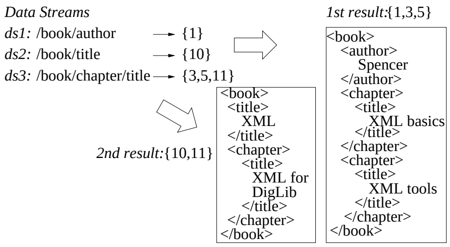


Fig. 5. Query embedding in CI


Data Streams

ds1: /book/author → {1}

ds2: /book/title → {10}

ds3: /book/chapter/title → {3,5,11}

1st result:{1,3,5}

```
<book>
  <author>
    Spencer
  </author>
  <chapter>
    <title>
      XML basics
    </title>
  </chapter>
  <chapter>
    <title>
      XML tools
    </title>
  </chapter>
</book>
```

2nd result:{10,11}

```
<book>
  <title>
    XML
  </title>
  <chapter>
    <title>
      XML for
      DigLib
    </title>
  </chapter>
</book>
```

Fig. 6. Sample query results

the leaves of the query tree[15] to retrieve the data that satisfies the corresponding query semantic conditions, i.e. data that contains the keyword(s) specified in the query. We call each list of relevant data leaves returned by a value-index a *data stream*. For instance, in Fig. 6 three data streams are shown: One for each match of the query (tree) leaves in the index. As an example, the data stream $ds2$ is a list made of a single element, the data leaf numbered 10: In fact, as shown in Fig. 2, only leaf 10 is relevant to the query path $/b/t/``XML''$. This retrieval is performed by the value-index assigned to the index path $/b/t$. Each stream denotes a different data source according to different structural approximations. For instance, in Fig. 5, the data streams assigned to index paths $/b/t$ and $/b/c/t$ represent data sources assigned to exact match and approximate match of query path $/b/t/``XML''$, respectively.

However, the assignment of data streams does not suffice. Although each data stream returns relevant data leaves, we have to verify that they belong to the same (sub)document instance. As an example, according to the document collection shown in Fig. 2, only document $dt1$ is relevant to the query shown in Fig. 6, since "Spencer" appears as author only in $dt1$. This implies that only data leaves numbered 1, 3, and 5 compose a relevant result. On the other hand, data leaves numbered 10 and 11 belong to an incomplete result that does not satisfy condition on author.[16] In order to assert which data of n data streams belongs to the same relevant result, we take advantage of the list ρ of ranges assigned to index inner nodes.

Construction of results proceeds according to a post-order visit of the extended query tree obtained at step 1), exploiting additional information which is recursively attached to each query node. In fact, at the end of the construction process, each node of the query tree is assigned a list of *contextual ranges*, i.e. a list of pairs `(context,Ranges)`, such that `Ranges` is a list of ranges of relevant data leaves, and `context` is the context in the index where these have been retrieved. For instance, in Fig. 5 query node b is assigned the pair `(index_root,`$\langle$`[1,7]`$\rangle$`)`, since relevant data leaves 1, 3, and 5 that compose the first result shown in Fig. 6 belong to range [1,7]. This implies that, the *did* as-

[15] Recall that we refer to *data leaves* as CDATA/PCDATA content, and to *(tree) leaves* as tree nodes that refer to data leaves.

[16] The last incomplete result will be taken into account in Section 4.3 when dealing with approximate embeddings allowing renaming and deletion of query nodes.

```
FindEmbeddings(IndexTree I, QueryTree q)
begin
1.  if leaf(q)
2.    for each m=(context,match) $\in$ list of contextual matches of q
3.        Ranges $\leftarrow$ getDataStream(I,match) s.t. semantic condition of q is satisfied
4.        append r=(context,Ranges) to list of contextual ranges of q
5.  else
6.    for each $q_j$ $\in$ children(q)
7.        FindEmbeddings(I,$q_j$)
8.    ChildrenLists $\leftarrow$ get lists of contextual ranges from children of q
9.    ChildrenList $\leftarrow$ merge ChildrenLists per context
10.   IndexList $\leftarrow$ $\emptyset$
11.   for each m=(context,match) $\in$ list of contextual matches of q
12.       append (m,getRangeList(I,match)) to IndexList
13.   RangeList $\leftarrow$ InclusionCheck(ChildrenList,IndexList)
14.   assign RangeList as list of contextual ranges of q
end
```

Fig. 7. Algorithm `FindEmbeddings`

signed to range [1,7] in the ρ list of index node b is the id of the document that satisfies the query. For a query (tree) leaf l, the list `Ranges` correspond to a list of data streams, one for each match of l in the index. The list of contextual ranges for a query node n, each denoting a (sub)query embedding is constructed recursively, according to Algorithm `FindEmbeddings`:

1. For each contextual match of a query leaf l, the data stream of the match is assigned to l as list of ranges of relevant data leaves in the context of the match. This is the basic case of single-element ranges, one per relevant data leaf (lines 2 – 4).
2. As to each inner query node n, the algorithm proceeds recursively to generate a list of contextual ranges for each child node of n (lines 6 – 7). Then, all lists of contextual ranges are collected from children nodes of n (line 8). Contextual ranges that share the same context c are merged in order to collect all relevant data leaves in a single occurrence of context c (line 9). Thus, `ChildrenList` contains the list of contextual ranges referring relevant data leaves that satisfy conditions specified by children nodes of n. Recall that these ranges have to be aggregated according to their membership to each (sub)document instance. This is accomplished by taking advantage of the ρ lists of the contextual matches of n, which are collected in the list `IndexList` (lines 10 – 12). Recall that each range in `IndexList` references a (sub)document instance, and that ranges of relevant data leaves in `ChildrenList` belong to the same (sub)document instance d_i if they are included in the range r in `IndexList` that refers d_i. This implies to check inclusion of ranges of `ChildrenList` into ranges of `IndexList` (line 13). Ranges in `IndexList` that satisfy inclusion are assigned as contextual ranges of node n. For each of these ranges, each context is given by the context assigned to the corresponding match of n in the index.

In order to guarantee that all query conditions are satisfied, the check inclusion step requires that at least one data leaf per query path belongs to the same

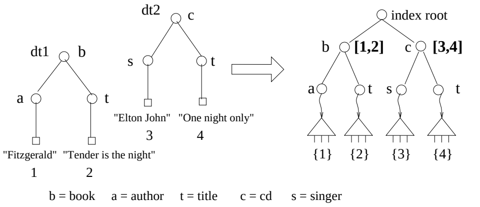


Fig. 8. Heterogeneous documents in CI

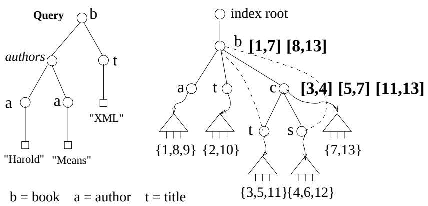


Fig. 9. Sample query on the CI

contextual range (*completeness check*).[17] At the end of the construction process, the contextual ranges assigned to the root of the query denote the (sub)document instances that satisfy the query conditions. In our example of Fig. 6, only the first result is returned. Approximation is due to the presence of the `chapter` element between the `book` and `title` in the query path */book/title/"XML"*. This result complies with the definition of tree embedding. Element `book` is the root of two query embeddings: The former matching book's author and the title of first chapter; The latter matching book's author and the title of second chapter. This implies that the algorithm `FindEmbeddings` finds out several occurrences of the same pattern in a document.

4.2 Complexity of Query Processing

The basic feature of the index is the list of labelled descendants (δ) for each inner node $n \in \mathcal{N}_I$. Efficiency and effectiveness of of top-down navigation of the Collection Index are improved: Query execution is lightened, since intermediate nodes leading to each descendant node are not visited; On the other hand, the navigation strategy leads to exploring only *relevant contexts*, a priori excluding parts of the index tree denoting different scopes. As an example, consider Fig. 8 and query */book/title/"night"*, looking for books having the keyword "night" in the title. The navigation step looking for element `book` reduces the scope of the search to the left part of the Collection Index, whereas approaches "à la ApproXQL" check for relevant data leaves also for titles of CD's.

Recalling Section 2, the complexity of performing approximate complex queries with the Collection Index is $O(n_q{*}l{*}s_c^I{*}p^I)$. More precisely, this is given by the sum of two quantities: $O(n_q{*}l{*}s_c^I) + O(n_q{*}l{*}s_c^I{*}p^I)$. Quantity $O(n_q{*}l{*}s_c^I)$ is due to Index navigation to set matches for query nodes (Algorithm `SetMatches`): In fact, for each query node at most s_c^I matches exist in all the *contexts* of the parent node, and all of these can be found at most as descendants of l contexts. Quantity $O(n_q{*}l{*}s_c^I{*}p^I)$ is due to construction of results (Algorithm `FindEmbeddings`): For each query leaf, the number of contextual ranges is at most $l{*}s_c^I$, one for each contextual match. For each inner query node, the merge step (line 9) requires $O(l{*}s_c^I)$ operations, since lists are ordered per context,[18] and $l{*}s_c^I$ is the maximal length of a list of contextual ranges. For each pair of contextual ranges

[17] We will relax this step in Section 4.3 to capture also partial embeddings.

[18] Recall that the δ list of each index node contains pointers to descendants obtained by a pre-order visit of the Index. This implies that the list of contextual matches (obtained from the δ list, in Algorithm `SetMatches`) is ordered according to this

merged at this step, the corresponding lists of ranges are to be collected as a single list per context. This additional merging step requires $O(p^I)$ operations since lists of ranges are ordered[19], and each list contains at most p^I elements. Thus, operation at line 9 costs $O(l{*}s_c^I{*}p^I)$. After this merging step, `ChildrenList` contains at most s_c^I contextual ranges, each one having a range list of length at most $fo_{max}{*}p^I$, where fo_{max} is the maximal fan-out of nodes in the Collection Index. `IndexList` contains at most $l{*}s_c^I$ ordered elements, each referring $O(p^I)$ ordered ranges (lines $10-12$). Thus, the inclusion check step for ranges (line 13) costs $O(l{*}s_c^I{*}p^I)$, since lists are ordered. In conclusion, Algorithm `FindEmbeddings`, which dominates the overall complexity, can be executed in time $O(n_q{*}l{*}s_c^I{*}p^I)$.

4.3 Retrieving all Approximate Embeddings

For the sake of simplicity, in Section 4 we dealt with approximate embeddings that relax query conditions only to insertion of nodes in the data. Now we consider two additional kinds of approximations: *Renaming* and *deletion* of query nodes. Consider the query shown in Fig. 5, where label `writer` is set in place of `author`. Results would be empty since no index node exists having label `writer`. Renaming of query nodes can be easily supported by the Collection Index: Traversal of arcs can be relaxed to descendants having *similar* labels. This allows for *semantic approximation*. Similarity can be measured as a semantic relationship between node labels.[20] Consider again query in Fig. 5, with name of the author changed to “Brown”. No document in our sample data collection satisfies all query conditions. Following a more flexible approach, relaxation on completeness of results would produce both (approximate) results shown in Fig. 6. However, this is not the only relaxation that can be made as to query partial matching, since the removal of the completeness check step corresponds to deletion of query data leaves. Consider Fig. 9: results are empty because label `authors` does not appear in the Index. To overcome this drawback, the navigation strategy (Algorithm `SetMatches`) can be modified: a query node that does not find any match in the index should be marked as *unsatisfied*, and navigation should proceed with next query node.[21]

Note that the last navigation strategy leads to *controlled* structural approximations. In fact, a query node is allowed to be deleted only if it does not find a match in the index. This implies that only the *structurally best-matching* results are returned. As an example, consider the query $/book/chapter/title/$“$DigLib''$. In Fig. 2 the query pattern occurs in document $dt2$, which is relevant to the query. This is the only result captured by the navigation strategy proposed, that does not allow query node `chapter` to be deleted, since at least one structural occurrence $/book/chapter$ is present in the collection. On the other hand, ApproXQL retrieves a larger set of approximate embeddings, namely those obtainable by the deletion of any query node, except the root. According to this

visit. As a consequence, elements in each list of contextual ranges (obtained at line 2 of Algorithm `FindEmbeddings`) respect the same order.

[19] As discussed in Section 3.2.

[20] To this end, in our implementation we relied on the WordNet semantic network [3].

[21] ApproXQL only allows for deletion of nodes having at most one child.

approach, also document $dt1$ is relevant to the above query, since it is a book having in the title the searched keyword. Of course, also document $dt1$ might be of interest for the user. This approximation can be obtained by means of an additional relaxation of the navigation strategy: At each navigation step all arcs leading to descendants having a label among the labels of the query pattern are traversed. Thus, depending on the query constraints the user may want to set to possible approximations on query results, a different navigation strategy can be applied on the Collection Index. It is worth to note that, the retrieval of all approximate embeddings according to the deletion of any query node, on one hand guarantees to obtain all relevant data, on the other hand, it may generate a huge amount of results that satisfy the query pattern only minimally, thus lowering the precision of the result set.

Ranking Results The relevance of results takes into account several aspects. According to the $SATES$ proposal [6, 7], we compute the score of results as a combination of semantic and structural similarities, providing a ranking according to correctness, completeness, and cohesion of the data retrieved.

5 Experiments

We used the XMach collection [16] for testing our index. The collection presents some interesting characteristics that emphasize the importance of providing approximate query capabilities: The documents have *irregular* structure and a high nesting level (up to 13), with label repetition along a path which is at most 11. In such a structural scenario, the recall of an exact match approach would not be satisfactory. XML files in the collection respect a DTD[22] that models documents having an author, a title and 1 or more chapters, each one with usual information, like author, sections, etc. We experienced the Collection Index on XMach subcollections ranging from 1,000 to 150,000 documents, with total size ranging from 16Mb to 2.3Gb, respectively. The index scales linearly in space, and for large collections exceeds the original size of the dataset by 1,5-2%. As to time scalability, the Collection Index proved to perform linearly with the size of the dataset.

6 Conclusion and Future Work

We have presented the Collection Index, an indexing method that efficiently supports the retrieval of approximate results for complex structured queries on large collections of XML documents. To authors' knowledge it is the first index that supports complex branching queries with both content and structural approximations, also capturing the presence of *multiple occurrences* of query conditions. A similar approach having the same goals has been presented in [15]. The ApproXQL system copes with the intrinsic complexity of finding unordered tree embeddings [10], and proposes a method that solves a reformulation of the classical problem in polinomial time. We proved the Collection Index to support a more efficient query processing, through the selection of *relevant data contexts*.

[22] We remind that the Collection Index does not require the presence of a DTD.

Results are *ranked* according to an effective measure that takes into account semantic and structural correctness, completeness, and cohesion of results [6, 7].

The Collection Index seems to be an effective and flexible indexing structure, where different strategies can be applied to personalize the user needs. Studying which strategies may prune less relevant results, with the aim of scaling down further the complexity of finding approximate embeddings for a query tree is future work. Then, we plan to exploit the numbering scheme applied to the data leaves, to support queries with ordering requirements. Query processing optimization is a further direction we intend to follow, by studying the evaluation ordering of structural predicates to improve index performance.

References

1. ACM Sigmod Online. `http://www.acm.org/sigmod/record/xml/`.
2. DBLP XML records. `http://www.informatik.uni-trier.de/ ley/db/index.html`.
3. WordNet Home Page. `http://www.cogsci.princeton.edu/ wn/`.
4. XML Information Set. `http://www.w3.org/TR/xml-infoset`.
5. S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the 8th Int. Conf. on Extending Database Technology (EDBT 2002)*, March 2002.
6. P. Ciaccia and W. Penzo. Adding Flexibility to Structure Similarity Queries on XML Data. In *Proc. of 5th Int. FQAS Conf. (2002) LNAI 2522*, pages 124–139.
7. P. Ciaccia and W. Penzo. Relevance Ranking Tuning for Similarity Queries on XML Data. In *Proc. of the 1st VLDB Workshop EEXTT 2002*, China, 2002.
8. B. F. Cooper, N. Sample, M. J. Franklin, and M. Shadmon G. R. Hjaltason. A Fast Index for Semistructured Data. In *Proc. of the 27th VLDB Conf.*, 2001.
9. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of 23rd Int. VLDB Conf.*, 1997.
10. P. Kilpeläinen. *Tree Matching Problems with Application to Structured Text Databases.* PhD thesis, Dept. of Computer Science, Univ. of Helsinki, SF, 1992.
11. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th VLDB Conf.*, pages 361–370, Rome, Italy, 2001.
12. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of the 8th Int. Conf. on Database Theory (ICDT'99)*, pages 277–295, Jerusalem, Israel, 1999.
13. F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *Proc. of 4th Int. Work. on the Web and Databases (WebDB01)*, 2001.
14. T. Schlieder. Similarity Search in XML Data Using Cost-Based Query Transformations. In *Proc. of 4th Int. Work. on the Web and Databases (WebDB01)*, 2001.
15. T. Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In *Proc. of the 8th Int. EDBT Conf.*, 2002.
16. E. Rahm T. Böeme. XMach-1: A Benchmark for XML Data Management. In *Proc. of Conf. on Database Systems for Business, Technology ad Web (BTW 2001).*
17. A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Proc. of the 8th Int. EDBT Conf.*, 2002.