Similarity Search in XML Data using Cost-Based Query Transformations

Torsten Schlieder*

Freie Universität Berlin

schlied@inf.fu-berlin.de

Abstract

XML query engines should support structured queries. They should retrieve exact matches as well as results similar to the query. In this paper, we introduce the simple query language approXQL that supports hierarchical, Boolean-connected query patterns. The interpretation of approXQL queries is founded on cost-based query transformations: The total cost of a sequence of transformations measures the similarity between a query and the data and is used to rank the results. All results of an approXQL query can be computed in polynomial time with respect to the database size.

1 Introduction

An XML query engine should retrieve the best results possible. If no exactly matching documents are found, results *similar* to the query should be retrieved and *ranked* according to their similarity. The problem of similarity between keyword queries and text documents has been investigated for years in information retrieval [BR99]. Unfortunately, the models developed in this community cannot be directly applied to XML documents, since they

- ignore the structure of XML documents and therefore lower the retrieval precision, and
- use models based on term distribution that are of little use for *data-centric* XML documents.

XML query languages, on the other hand, do incorporate the document structure. They are well suited for applications that query and transform XML documents [BC00]. However, they do not well support user queries, since

- results that do not fully match the query are not retrieved, and
- the user needs a thorough knowledge of the data structure to formulate queries.

Assume a catalog storing data about sound storage media. A user may be interested in a CD with piano concertos by Rachmaninov. A keyword query retrieves all documents that contain at least one of the terms "piano", "concerto", and "Rachmaninov". However, the user cannot specify that she *prefers* CDs with the title "piano concerto" over CDs containing a track title "piano concerto". Similarly, the user cannot prefer the composer Rachmaninov over the performer Rachmaninov.

Structured queries yield the contrary result: Only exactly matching documents are retrieved. The XQL [RLS98] query

```
/catalog/cd[composer="Rachmaninov" $and$ title="Piano concerto"]
```

will neither find CDs with a *track* title "Piano concerto" nor CDs of the *category* "Piano concerto" nor concertos *performed* by "Rachmaninov", nor other sound storage media than CDs with the appropriate information. The query will also not find CDs where only one of the specified keywords appears in the title. Of course, the user can pose a query that exactly matches the cases mentioned – but she must know

*This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

beforehand that such similar results exist and how they are represented. Moreover, since all results of the expanded query are treated equally, the user still cannot express her preferences.

As a first step to bridge the gap between the vagueness of information retrieval and the expressiveness of structured queries with respect to data-centric documents, we introduce the simple pattern matching language approXQL. The interpretation of approXQL queries is founded on cost-based query transformations. The total cost of a sequence of transformations measures the similarity between a query and the data. The similarity score is used to *rank* the results. We allow renaming of query elements, insertion of elements into the query, and deletion of a restricted set of query elements. Each type of transformation has its intuitive semantics: The renaming of a query element *changes* the search context of a query part. The insertion of a query element restricts a query part to a *more specific* context. Finally, the deletion of a query element changes the search space to a *more general* context. The costs of the basic transformations renaming, insertion, and deletion are specified by a domain expert.

All results of an approXQL query can be computed in polynomial – typically sublinear – time with respect to the database size. This favorable time complexity and the XML tailored query semantics are the main differences to well-known tree similarity measures [AG97].

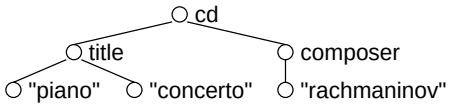
Only few other papers investigate result ranking for structured queries against XML documents: Fuhr and Großjohann [FG00] as well as Hayashi et. al. [HTK00] focus on *text-rich* documents, whereas our language is designed for *data-centric* documents. Theobald and Weikum [TW00] propose the query language XXL, which supports a similarity operator. To use this operator, the user must know that similar keywords or element names exist. In contrast, the approXQL query processor transforms the query automatically in order to find appropriate results. Unlike all approaches mentioned, approXQL can handle *partial structural matches*.

2 The ApproXQL Query Language

ApproXQL is a simple pattern matching language inspired by XQL [RLS98]. Informally, approXQL consists of (1) name selectors, (2) text selectors, (3) the containment operator "`[]`", and (4) the logical operators "`$and$`" and "`$or$`". The query depicted in Figure 1(a) selects CDs that contain piano concertos composed by Rachmaninov. Note, that the text selectors match both text data and attribute values. Any conjunctive approXQL query can be interpreted as a tree (see Figure 1(b)). Text selectors are mapped to leaf nodes of type *text*; name selectors are represented as *structural* nodes. Each `$and$` expression is mapped to an inner node of the tree. The children of an `$and$` node are the roots of the paths that are conjunctively connected.

```
cd[title["piano" $and$ "concerto"] $and$
   composer["rachmaninov"] ]
```

(a) Conjunctive approXQL query



(b) Query tree

Figure 1: Mapping of a conjunctive approXQL query to a tree.

Queries containing `$or$` operators are decomposed into a *disjunctive normal form*. The disjunctive normal form of a query is a set of conjunctive queries. For instance, the query

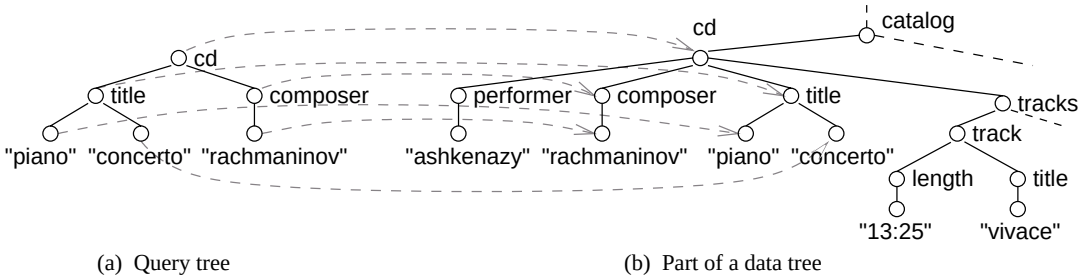
```
cd[year["2001"] $and$ (composer["rachmaninov"] $or$ performer["ashkenazy"])]
```

is decomposed into the set of conjunctive queries

```
{ cd[year["2001"] $and$ composer["rachmaninov"]],
  cd[year["2001"] $and$ performer["ashkenazy"]] }.
```

3 Modeling and Normalization of XML Documents

We model XML documents as labeled trees, neglecting the semantics of links. To simplify our model, we only use two node types: *structural* nodes represent elements and attributes; *text* nodes represent element text



(a) Query tree (b) Part of a data tree

Figure 2: Embedding of a query tree into a data tree.

as well as attribute values. The name of an element is used as node label. Text sequences are decomposed into words, which are stemmed and converted to lower case. For each word, a leaf node of the document tree is created and labeled with the word. Attributes are mapped to two nodes which are parent and child of each other: The attribute name forms the label of the parent, and the attribute value forms the label of the child. We add a new root node with a unique label to the collection of document trees and establish an edge between this node and the root of every document tree. The tree representing all document trees is called *data tree*.

4 Querying by Approximate Tree Matching

In this section, we introduce the semantics of evaluating an approXQL query. We first define an embedding function that maps a conjunctive query tree to a data tree. Then, we introduce our approach to find results similar to the query by inserting, renaming, and deleting query nodes.

An *exact embedding* is a function that maps the query nodes to the data nodes. The mapping function f has the following properties:

1. f is label preserving,
2. f is node-type preserving, and
3. f is parent-child preserving.

We call the data node that is mapped to the query root the *embedding root*. The data subtree that is anchored at the embedding root is a *result*. Note, that for a fixed query tree and a fixed data tree, several results may exist, and several embeddings may lead to the same result. Figure 2 shows an embedding of a query tree into a data tree.

Approximate tree embedding is based on the query transformations renaming, insertion, and deletion of nodes. In contrast to the tree edit distance [Tai79], we do not allow arbitrary sequences of basic transformations. The main purposes of these restrictions are (1) to generate only queries that are semantically useful, and (2) to avoid combinatorial effects that make the problem NP-hard [AG97].

Renaming: Renaming nodes means *changing the context* and thus changing the search space of a query subtree. For example, renaming the root of the query depicted in Figure 2(a) from cd to mc obviously changes the search space from CDs to MCs; the renaming of composer to performer changes the context in which the keyword "rachmaninov" is expected. Similarly, the renaming of the keyword "concerto" to "sonata" changes the context of the text selector.

Insertion: A node insertion creates a query that expects the matches of a query subtree in a *more specific context*. We only allow the insertion of inner nodes, that is, every edge can be replaced by a node with an incoming and an outgoing edge. For example, the insertion of nodes labeled tracks and track between the nodes cd and title of the query shown in Figure 2(a) creates a query that searches for subtree matches in the more specific context of track titles. Cost-based node insertions can be considered as a valuated variant of the "//" operator of XQL.

Deletion: The deletion of a query node moves the search space to a *more general context*. The deletion is founded on two concepts: vague containment and vague conjunction. Vague containment is based on the observation that the hierarchy of an XML document typically models a containment relationship. The deeper an element resides in the document tree the more specific is the information it describes. Therefore, the deletion rule of vague containment allows the deletion of inner query nodes *bottom up*. Vague conjunction adopts the concept of "coordination level match" [BR99], which is a simple querying model that establishes ranking for queries of `and`-connected search terms. In this model, documents containing all n terms of the query score higher than documents containing $n-1$ terms. Sequences of deleting inner nodes and leafs are not commutative. All deletions of inner nodes must precede all deletions of leafs. In the query tree shown in Figure 2(a), the inner nodes title and composer may be deleted. Afterwards, at most two of the three leafs of type *text* may be deleted.

Each basic transformation has a cost that is determined by the label(s) of the node(s) involved. The total cost of a sequence of transformations is called *embedding cost*. The following steps describe the principle of finding and ranking the results of an approXQL query:

1. Decompose the user query into its conjunctive normal form.
2. Successively create the "closure" of the set of conjunctive queries by adding new queries that are derived from existing ones by applying node insertions, deletions, and renamings.
3. Try to exactly embed all queries created this way into the data tree.
4. Group the embeddings according to the results they belong to.
5. From each group, choose the embedding with the lowest cost.
6. Rank the results according to their embedding cost.

5 A Polynomial Query Evaluation Algorithm

The approximate tree matching model, introduced in the previous section explicitly creates an (infinite) set of transformed queries from an original one. This model seems to require an implementation that has an exponential runtime complexity. Fortunately, as we will see in this section, the problem of finding all approximate results for a given query can be solved by an algorithm that has polynomial worst-case time complexity.

The basic idea is to represent all possible insertions in an encoded data tree, and to represent all allowed renamings and deletions in an expanded representation of the query. Such a representation is always feasible since the definitions of tree embedding, node insertion, deletion, and renaming avoid all combinatorial effects. The query evaluation algorithm adopts the principle of dynamic programming. It processes the expanded query bottom-up and computes the list of cost-minimal embedding roots of every query subtree. Consequently, the list belonging to the query root is retrieved as list of valuated results.

5.1 Encoding of the Data Tree, Partial Index, and Postings

We use an indexing technique that is inspired by the *partial index* introduced in [Nav95]. The indexes rely on a preorder enumeration of the data tree. Every text node u is represented by a pair $(pre(u), dist(u))$ consisting of the preorder number of u and the sum of the insert costs of all ancestors of u. Every structural node u is described by a triple $(pre(u), dist(u), bound(u))$, where $bound(u)$ refers to the preorder number of the rightmost leaf node of the tree rooted at u. Given two nodes u and v we can test if u is an ancestor of v by ensuring the invariant

$$pre(u) < pre(v) \wedge bound(u) \geq pre(v).$$

Let c be the cost of inserting u into a query. Then the cost of inserting all nodes along the path from u to v (excluding u and v) into a query is

$$dist(v) - dist(u) - c.$$

The indexes map node labels to postings. A posting P^x is either a list of pairs (text index), or a list of triples (structural index) that indicates all occurrences of nodes having label x. All postings are sorted by the preorder numbers in ascending order. Figure 3 shows an encoded data tree and its indexes. Here, all nodes have the insert cost 2.

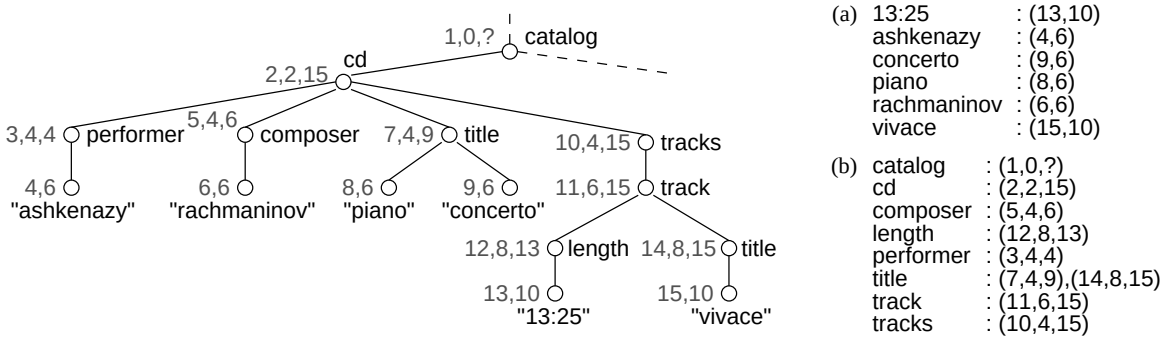


Figure 3: Encoded XML tree with its text index (a) and structural index (b).

5.2 Expanded Query Representation

The expanded representation of a query explicitly includes all deletions and renamings of query nodes that have a cost less than infinite. This representation captures the observation that, given a query and an embedding of the query into the data tree, the embedding cost of a query node only depends on its rename cost and on the embedding costs of its children. For example, the embedding cost of the title node of the query title["piano" and "concerto"] is the sum of the rename cost of the title node and the embedding costs of the leafs "piano" and "concerto". The embedding cost of the "concerto" leaf is the minimum of (1) its delete cost and (2) the sum of renaming the node and inserting nodes between title and "concerto". Vague containment prescribes that an inner query node can only be deleted if all its inner descendants are deleted. If v is a child of a query node u, and if v is not a leaf, then the part of u's embedding cost contributed by v is the minimum of (1) the distance between u and v plus the embedding cost of the query subtree rooted at v, and (2) the total cost of deleting v and all its inner descendants plus the embedding cost of the leafs.

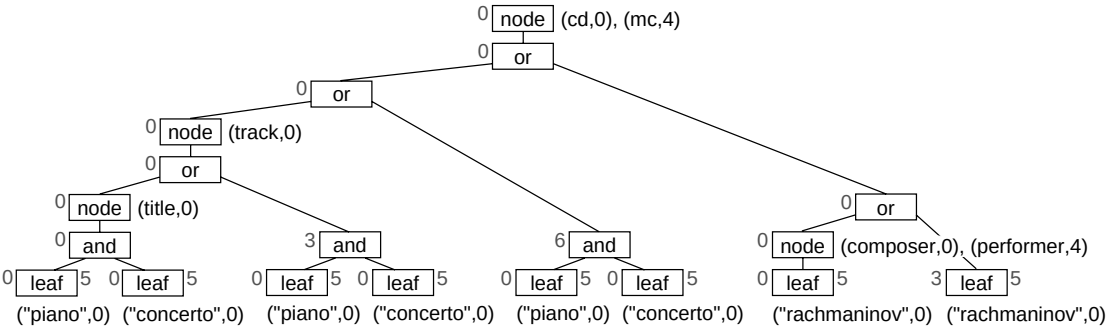


Figure 4: Expanded representation of the query
cd[track[title["piano" and "concerto"]] or composer["rachmaninov"]].

Figure 4 shows an expanded query representation. In this example query, the only inner nodes that may be deleted are title, composer, and track (after title has been deleted). They are represented by *or* nodes. The right child of each of these *or* nodes carries the total cost of deleting the query node and all its inner descendants. Assume, that the uniform cost of deleting an inner node is 3. This cost is assigned to the second left *and* node in Figure 4, because this node represents the deletion of the title node. Similarly, the total delete cost assigned to the third left *and* node is 6, since it represents the deletion of track and title. The value on the right side of every *leaf* node stands for the cost of deleting the leaf. In our example, the delete cost of every leaf is 5. Nodes of types *node* and *leaf* are annotated with a set of pairs each storing a label l and the cost of renaming the original label to l.

5.3 Algorithm

In this extended abstract we only give an informal description of the query evaluation algorithm, using the expanded query depicted in Figure 4 and the partial index of the data tree shown in Figure 3. The algorithm

processes the query bottom up. Starting at the leftmost inner node (title), it fetches the postings P^{title} from the structural index and P^{piano} from the text index. All nodes in P^{title} that have no descendant in P^{piano} are removed. For all remaining nodes in P^{title}, the algorithm searches for the minimal distance to any of its descendants in P^{piano}. Whenever this distance is larger than the delete cost of the leaf "piano", then the delete cost is kept instead. The same procedure is applied to the other leaf "concerto", and the embedding costs of both children are added for each node in P^{title}. The algorithm continues with the track node and joins the posting P^{track} with the posting P^{title} as described. Then, the right branch of the *or* node is evaluated. The postings of both *or* operands are merged in a way that for every node occurring in both postings, the minimum cost is chosen and updated in P^{track}. Finally, the algorithm reaches the query root and outputs the list of embedding roots. Each embedding root represents a result and the cost of the cheapest embedding belonging to the result.

The time complexity of the query evaluation algorithm is bound by $O(t \cdot n \cdot (i + r \cdot l))$. In this formula, t is the number of text selectors, and n is the number of name selectors of the original query. The letter i describes the time to access the index, r is the maximal number of alternative labels of a query node, and l is the number of entries of the largest posting. Since t, n and r are small numbers in practical cases, and every posting typically represents only a small fraction of the data nodes, we expect a sublinear average time complexity of the algorithm with respect to the database size. First experiments with our prototypical implementation indicate the correctness of our assumption.

6 Conclusion and Outlook

In this paper, we introduced an approach to find similar results to structured queries using cost-based query transformations. The query language and its implementation is explained in more detail in the extended version of this paper [Sch01]. There, we also present a method to find the best N query transformations at compile time using a DataGuide [GW97], language constructs to relax or enforce query transformations, and a description of our prototypical implementation.

We plan to test our approach in real-world experiments in order to verify its usefulness and to develop domain-specific rules for choosing basic transformation costs. Future work will also include the development of an adaptive user interface that provides a simplified view on the data structure and supports the user in formulating queries.

References

[AG97] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*, chapter 14. Approximate Tree Pattern Matching. Oxford University Press, June 1997.

[BC00] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1), March 2000.

[BR99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Longman, 1999.

[FG00] N. Fuhr and K. Großjohann. XIRQL: An extension of XQL for information retrieval. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured data. In *Proceedings of the 23rd International Conference on VLDB*, Athens, August 1997.

[HTK00] Y. Hayashi, J. Tomita, and G. Kikui. Searching text-rich XML documents with relevance ranking. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[Nav95] G. Navarro. A language for queries on structure and contents of textual databases. Master's thesis, Department of Computer Science, University of Chile, April 1995.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML query language (XQL), September 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[Sch01] T. Schlieder. ApproXQL: Design and implementation of an approximate pattern matching language for XML. Technical Report B 01-02, Freie Universität Berlin, May 2001.

[Tai79] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[TW00] A. Theobald and G. Weikum. Adding relevance to XML. In *Proceedings of 3rd International Workshop on the Web and Databases (WebDB'00)*, Dallas, USA, May 2000.