Schema-Driven Evaluation
of Approximate Tree-Pattern Queries

Torsten Schlieder⋆

Institute of Computer Science
Freie Universität Berlin
schlied@inf.fu-berlin.de

Abstract. We present a simple query language for XML, which supports hierarchical, Boolean-connected query patterns. The interpretation of a query is founded on cost-based query transformations: The total cost of a sequence of transformations measures the similarity between the query and the data and is used to rank the results. We introduce two polynomial-time algorithms that efficiently find the best n answers to the query: The first algorithm finds all approximate results, sorts them by increasing cost, and prunes the result list after the nth entry. The second algorithm uses a structural summary –the schema– of the database to estimate the best k transformed queries, which in turn are executed against the database. We compare both approaches and show that the schema-based evaluation outperforms the pruning approach for small values of n. The pruning strategy is the better choice if n is close to the total number of approximate results for the query.

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 [3]. Unfortunately, the most models (with some recent exceptions, e.g., [15,6,7]) consider unstructured text only and therefore miss the change to yield a more precise search. Furthermore, it is not clear whether retrieval models based on term distribution can be used for *data centric* documents as considered in this paper.

XML query languages, on the other hand, do incorporate the document structure. They are well suited for *applications* that query and transform XML documents [5]. However, they do not well support *user* queries because results that do not fully match the query are not retrieved. Moreover, the user needs substantial knowledge of the data structure to formulate queries.

Consider a catalog with 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 having a track title "piano concerto". Similarly, the user cannot express her preference for the composer Rachmaninov over the performer Rachmaninov.

Structured queries yield the contrary result: Only exactly matching documents are retrieved. The XQL [11] query

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

will neither retrieve 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 retrieve 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 beforehand that such similar results exist and how they are represented. Moreover, since all results of the redefined 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 present two polynomial-time algorithms that find the best n answers to the query: The first algorithm finds all approximate results, sorts them by increasing cost, and prunes the result list after the nth entry. The second algorithm is an extension of the first one. It uses the *schema* of the database to estimate the best k transformed queries, sorts them by cost, and executes them against the database to find the best n results. We discuss the results of experiments, which show that the schema-based query evaluation outperforms the pruning approach if n is smaller than the total number of approximate results.

2   Related Work

The semantics of our query language is related to cost-based distance measures for unordered labeled trees such as the tree-edit distance [14] and the tree-alignment distance [9]. Our approach is different concerning its semantics and concerning its computational complexity.

We believe that different nodes of a tree-shaped query pattern should be treated differently: Leaf nodes specify the *information* the user is looking for. The root node defines the *scope* of the search. The inner nodes determine the *context* in which the information should appear. None of the tree-similarity measures we know has a semantics tailored to XML data.

The problem of finding the minimal edit or alignment distance between unordered trees is MAX SNP-hard [2]. Even the problem of including a query tree into a data tree is NP-complete [10]. In [16] a restricted variant of the edit distance and its adaption to tree-pattern matching has been proposed. All matching subtrees can be found in polynomial time. The proposed algorithm touches every data node, which is inadequate for large databases.

To our knowledge, our work is the first in the context of approximate tree-pattern matching that proposes an *XML-tailored* query interpretation, supports *Boolean operators*, evaluates a query using *indexes* and *list operations*, and takes advantage of a *schema* to find the *best n* answers.

3   The ApproXQL Query Language

ApproXQL [12] is a simple pattern-matching language for XML. The syntactical subset of the language that we will use throughout the paper consists of (1) name selectors, (2) text selectors, (3) the containment operator "[]", and (4) the Boolean operators "and", "or". The following query selects CDs containing piano concertos composed by Rachmaninov:

```
cd[title["piano" and ''concerto"] and composer["rachmaninov"]].
```

Note that the text selectors match both text data and attribute values. A conjunctive query can be interpreted as a labeled, typed tree: Text selectors are mapped to leaf nodes of type *text*; name selectors are represented as nodes of type *struct*. 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. Figure 1(a) shows the tree interpretation of the above query.

A query that contains "or"-operators is broken up into a set of conjunctive queries, which is called *separated query representation*. The query

```
cd[title["piano" and ("concerto" or ''sonata")] and
   (composer["rachmaninov"] or performer["ashkenazy"])].
```

consists of two "or"-operators and can be converted into 2^2 conjunctive queries:

```
{ cd[title["piano" and ''concerto"] and composer["rachmaninov"]],
  cd[title["piano" and ''concerto"] and performer["ashkenazy"]],
  cd[title["piano" and ''sonata"] and composer["rachmaninov"]],
  cd[title["piano" and ''sonata"] and performer["ashkenazy"]] }.
```

4   Modeling and Normalization of XML Documents

We model XML documents as labeled trees consisting of two node types: *text* nodes represent element text as well as attribute values; nodes of type *struct* represent elements and attributes. The name of an element is used as node label. Text sequences are splitted into words. For each word, a leaf node of the
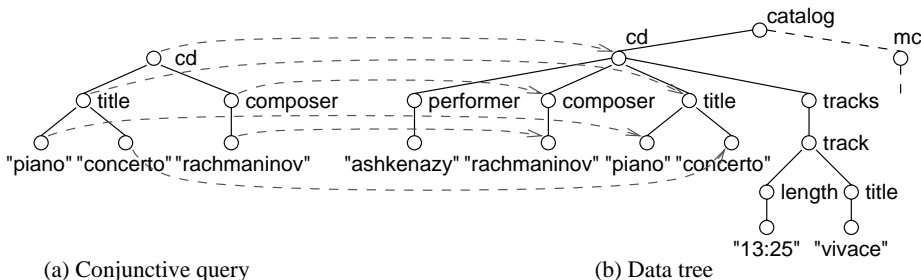
(a) Conjunctive query   (b) Data tree

Fig. 1. Embedding of a conjunctive query in a data tree.

document tree is created and labeled with the word. Attributes are mapped to
two nodes in parent-child relationship: 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 roots of the document trees. The resulting
tree is called *data tree*. Figure 1(b) shows a part of a data tree.

5   Querying by Approximate Tree Embedding

In this section, we introduce the semantics of approXQL queries. We first de-
fine an embedding function that maps a conjunctive query to a data tree. The
embedding is *exact* in the sense that all labels of the query occur in the result,
and that the parent-child relationships of the query are preserved. Then, we
introduce our approach to find *similar* results to a query.

5.1   The Tree-Embedding Formalism

Our definition of tree embedding is inspired by the *unordered path inclusion
problem* proposed by Kilpeläinen [10]. We discard the injectivity property of the
path inclusion problem in order to get a function that is efficiently computable[1]:

Definition 1 (Embedding). *An embedding of a conjunctive query into a data
tree is a function f that maps the query nodes to data nodes such that f is (1)
label preserving, (2) type preserving, and (3) parent-child preserving.*

Let u be a node of a conjunctive query. We call the data node $f(u)$ a *match* of u.
The match of the query root is the *embedding root* of f; the matched data nodes
together with the connecting edges are called *embedding image*; and the data
subtree anchored at the embedding root is a *result*. Note, that for a fixed query

---

[1] The injectivity of the embedding function together with the implicit relaxation of the
parent-child-relationship to an ancestor-descendant relationship (Section 5.2) would
lead to the *unordered tree inclusion problem*, which is NP-complete [10].

and a fixed data tree, several results may exist, and several embeddings may lead to the same result. Figure 1 shows an embedding of a conjunctive query into a data tree. The result of this embedding is the subtree rooted at the left cd node; all nodes with incoming arrows, together with their connecting edges, form the embedding image.

5.2   Basic Query Transformations

The tree-embedding formalism allows exact embeddings only. To find similar results to the query, we use *basic query transformations*. A basic query transformation is a modification of a conjunctive query by inserting a node, deleting a node, or renaming the label of a node. In contrast to the tree-edit distance [14], our model does not allow arbitrary sequences of insert, delete, and rename operations. We restrict the basic transformations in order to generate only queries that have intuitive semantics. For example, it is not allowed to delete *all* leaves of the original query, since every leaf captures information the user is looking for.

Definition 2 (Insertion). *An insertion is the replacement of an edge by a node that has an incoming edge and an outgoing edge.*

Note that this definition does not allow to add a new query root or to append new leaves. A node insertion creates a query that finds matches in a *more specific context*. As an example, consider the insertion of two nodes labeled tracks and track, respectively, between the nodes cd and title in the query shown in Figure 1(a). The insertions create a query that searches for subtree matches in the more specific context of track titles.

Definition 3 (Deletion of inner nodes). *A deletion removes an inner node u (except the root) together with its incoming edge and connects the outgoing edges of u with the parent of u.*

The deletion of inner nodes is based on the observation that the hierarchy of an XML document typically models a containment relationship. The deeper an element resides in the data tree the more specific is the information it describes. Assume that a user searches for CD tracks with the title "concerto". The deletion of the node track creates a query that searches the term "concerto" in CD titles instead of track titles.

Definition 4 (Deletion of leaves). *A deletion removes a leaf u together with its incoming edge iff the parent of u has two or more children (including u) that are leaves of the query.*

The deletion of leaves adopts the concept of "coordination level match" [3], which is a simple querying model that establishes ranking for queries of "and"-connected search terms.

Definition 5 (Renaming). *A renaming changes the label of a node.*

A renaming of a node u changes the search space of the query subtree rooted at u. For example, the renaming of the query root from cd to mc shifts the search space from CDs to MCs.

Each basic transformation has a cost, which is specified, for example, by a domain expert.

Definition 6 (Cost). *The cost of a transformation is a non-negative number.*

There are several variants to assign costs to transformations. In this paper we choose the simplest one: We bind the costs to the labels of the involved nodes.

5.3   The Approximate Query-Matching Problem

In this subsection, we define the approximate query-matching problem. We first define the terms *transformed query* and *embedding cost*:

Definition 7 (Transformed query). *A transformed query is derived from a conjunctive query using a sequence of basic transformations such that all deletions precede all renamings and all renamings precede all insertions.*

Each conjunctive query in the separated representation of an approXQL query is also a transformed query, which is derived by an empty transformation sequence.

Definition 8 (Embedding cost). *The embedding cost of a transformed query is the sum of the costs of all applied basic transformations.*

To evaluate an approXQL query, the *closure* of transformed queries is created from the separated query representation:

Definition 9 (Query closure). *The closure of a query Q is the set of all transformed queries that can be derived from the separated representation of Q.*

Every query in the closure of Q is executed against the data tree. Executing a query means finding a (possibly empty) set of embeddings of the query tree in the data tree according to Definition 1. All embeddings that have the same root are collected in an embedding group:

Definition 10 (Embedding group). *An embedding group is a set of pairs, where each pair consists of an embedding and its cost. All embeddings in a group have the same root.*

As an example, consider the query shown in Figure 1 and assume a further query that has been derived from the depicted query by deleting the node "concerto". Both queries have an embedding in the data subtree rooted at the left cd node. Therefore, both embeddings belong to the same embedding group. To get a single score for each group, we choose the embedding with the lowest embedding cost:

Definition 11 (Approximate query-matching problem). *Given a data tree and the closure of a query, locate all embedding groups and represent each group by a pair (u, c), where u is the root of the embeddings in the group and c is the lowest cost of all embeddings in the group.*

We call the pair (u, c) a *root-cost pair*. Each root-cost pair represents a result of the query. An algorithm solving the approximate query-matching problem must find *all* results of query. Since a user is typically interested in the *best* results only, we define the best-n-pairs problem as follows:

Definition 12 (Best-n-pairs problem). *Create a cost-sorted list of the n root-cost pairs that have the lowest embedding costs among all root-cost pairs for a query and a data tree.*

The following steps summarize the evaluation of an approXQL query:

1. Break up the query into its separated representation.
2. Derive the closure of transformed queries from the separated representation.
3. Find all embeddings of any transformed query in the data tree.
4. Divide the embeddings into embedding groups and create the root-cost pairs.
5. Retrieve the best n root-cost pairs.

In an additional step, the results (subtrees of the data tree) belonging to the embedding roots are selected and retrieved to the user. The five steps describe the evaluation of an approXQL query from the theoretical point of view. In the following sections we give a more practicable approach to evaluate a query.

6   Direct Query Evaluation

The approximate tree-matching model explicitly creates a (possibly infinite) set of transformed queries from a user-provided query. In this section, we show that the explicit creation of transformed queries is not necessary. Moreover, we show that the images of all approximate embeddings of a query can be found in polynomial time with respect to the number of nodes of the data tree. The evaluation of a query is based on three ideas: First, we encode all allowed renamings and deletions of query nodes in an *expanded representation* of the query. The expanded representation implicitly includes all so-called *semi-transformed* queries. Second, we detect all possible insertions of query nodes using a special numbering of the nodes in the data tree. Third, we simultaneously compute all embedding images of the semi-transformed query using a bottom-up algorithm. In the examples used in this section we assume the following costs:

insertion	cost	deletion	cost	renaming	cost
category	4	composer	7	cd → dvd	6
cd	2	"concerto"	6	cd → mc	4
composer	5	"piano"	8	composer → performer	4
performer	5	title	5	"concerto" → "sonata"	3
title	3	track	3	title → category	4

All delete and rename costs not listed in the table are infinite; all remaining insert costs are 1.

(a) Expanded representation   (b) Four semi-transformed queries
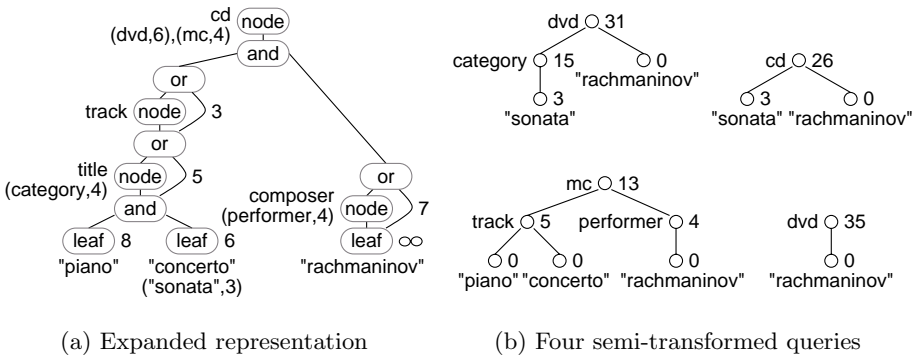
Fig. 2. Expanded representation and semi-transformed queries derived from the query `cd[track[title["piano" and ''concerto"]] and composer["rachmaninov"]]`.

6.1   The Expanded Representation of a Query

Many transformed queries in the closure of an approXQL query are similar; they often differ in some inserted nodes only. We call a query that is derived from a conjunctive query using a sequence of deletions and renamings (but no insertions) a *semi-transformed query*. The *expanded representation* of a query Q encodes all distinct semi-transformed queries that can be derived from the separated representation of Q. It consists of nodes belonging to four *representation types*:

node: A node of representation type "node" represents all nodes of all semi-transformed queries that are derived from the same inner node of the original query. Consider Figure 2(a). The top-level node represents the cd node of the original query and its renamings dvd and mc that have the costs 6 and 4, respectively.

leaf: A "leaf" represents all leaves of all semi-transformed queries derived from the same leaf of the original query. The middle leaf of the query in Figure 2(a) represents the "concerto" node of the original query. It is labeled with the original term and its single renaming "sonata", which has cost 3. Assigned to the right side of the leaf is the delete cost 6 of the node.

and: Any "and"-node represents an "and"-operator of the original query.

or: Nodes of type "or" have two applications: First, they represent "or"-operators of the original query. Second, for each inner node that may be deleted, an "or" node is inserted in the expanded query representation. The left edge leads to the node that may be deleted. The right edge bridges the node. It is annotated with the delete cost of the bridged node. In our example, every inner node (except the root) may be deleted and has therefore an "or"-parent.

A semi-transformed query can be derived from the expanded representation by following a combination of paths from the root to the leaves. The total cost of the

derived query consists of the rename cost of the choosen labels, the costs assigned to the edges and the costs of the deleted leaves. Figure 2(b) depicts four out of 84 semi-transformed queries included in the expanded query representation shown in Figure 2(a). The number assigned to each node represents the *minimal* cost of approximate embeddings of the subtree rooted at the node. Node insertions in the subtree may increase the costs.

We define a number of attributes for each node u of an expanded query representation: $reptype(u)$ is the representation type of u (and, or, node, leaf), $label(u)$ is the label, and $type(u)$ is the node type of u (*struct*, *text*). For each "node" and "leaf" the set $renamings(u)$ contains all alternative label-cost pairs for u, and $delcost(u)$ is the cost of deleting u. If u is an "or" node then $edgecost(u)$ denotes the cost assigned to the edge leading to the right child of u.

6.2   Encoding of the Data Tree

The embedding of a (transformed) conjunctive query into a data tree is defined as function that preserves labels, types, and parent-child relationships. In order to construct an embeddable query, nodes must be inserted into the query. This "blind" insertion of nodes creates many queries that have no embedding at all. We completely avoid the insertion of nodes into a query. Instead, we use a encoding of the data tree in order to determine the *distance* between the matches of two query nodes. More precisely, we change property (3) of the embedding function (see Definition 1) from "parent-child preserving" to "ancestor-descendant preserving" and define the distance between two nodes u and v as the sum of the insert costs of all nodes along the path from u to v (excluding u and v).

We assign four numbers to each data node u: $pre(u)$ is the preorder number of u; $bound(u)$ is the number of the rightmost leaf of the subtree rooted at u; $inscost(u)$ is the cost of inserting u into a query; and $pathcost(u)$ is the sum of the insert costs of all ancestors of u. Given two nodes u and v we can now test if u is an ancestor of v by ensuring the invariant

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

If u is an ancestor of v then the distance between u and v is

$$distance(u, v) = pathcost(v) - pathcost(u) - inscost(u).$$

An example of an encoded data tree is shown in Figure 3(a). The preorder number and the bound value are assigned to left side of each node; the pathcost value and the insert cost are located at the right side. We know that node 15 ("vivace") is a descendant of node 10 (tracks) because $10 < 15 \wedge 15 \geq 15$ evaluates to true. Using the expression $9 - 3 - 2 = 4$ we can determine the sum of the insert costs of the nodes 11 and 14 and thus, the distance between the nodes 10 and 15.

The indexes I_{struct} and I_{text} provide access to the nodes of the data tree by mapping each label to all nodes that carry the label. The Figures 3(b) and 3(c) show the indexes of the encoded data tree depicted in Figure 3(a).
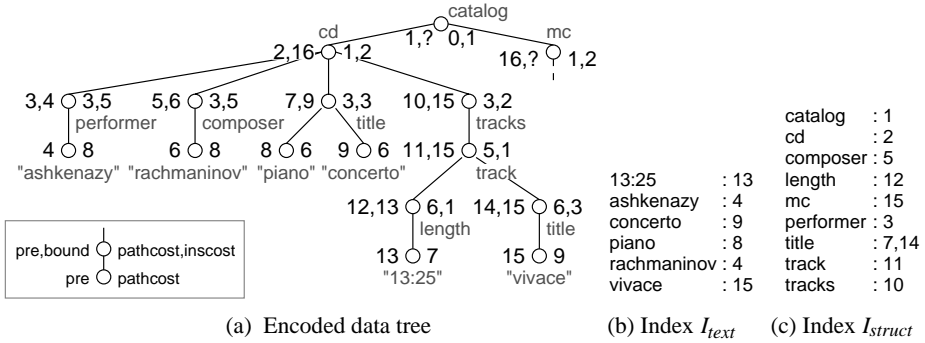
(a) Encoded data tree   (b) Index I_{text} (c) Index I_{struct}

Fig. 3. An encoded data tree with its text index and structural index.

6.3 Lists and List Entries

The query-evaluation algorithm computes all approximate embeddings using an algebra of lists. A *list* stores information about all nodes of a given label and is initialized from the corresponding index posting. A list entry e is a tuple

$$e = (pre, bound, pathcost, inscost, embcost),$$

where the first four values are copies of the numbers assigned to the correspond-ing node u. If u is a *text* node then *bound* and *inscost* are set to zero. All operations on lists are based on these four numbers. In particular, they serve to test the ancestor-descendant relationship and to compute the distance between two nodes. The value *embcost* stores the cost of embedding a query subtree into the data subtree rooted at u. The value is zero if u is the match of a query leaf. For convenience, we use the set notation $e \in L$ to refer to an entry of L.

6.4 Operations on Lists

We now introduce the basic list operations used by our query-evaluation algo-rithm. List operations are realized as functions that essentially perform standard transformations of lists but additionally calculate the embedding costs during the bottom-up query evaluation. The function `join`, for example, assumes that the embedding cost of each descendant $e_D \in L_D$ has already been calculated. The embedding cost of an ancestor $e_A \in L_A$ of e_D is therefore $distance(e_A, e_D) + embcost(e_D)$. Because e_A may have several descendants $e_{D_1}, \ldots, e_{D_m}$, we choose the one with the smallest sum of embedding cost and distance:

$$embcost(e_A) = \min\{ distance(e_A, e_{D_i}) + embcost(e_{D_i}) \mid 1 \leq i \leq m \}.$$

The cost is increased by the c_{edge}, which represents the cost of a deleted query node. We use the same principle for the function `intersect`, which calculates the sums of the embedding costs of corresponding entries in the operand lists,

for the function `union`, which chooses the lowest embedding costs of each pair of entries in the operand lists, and for the function `outerjoin`, which keeps the minimum of the cheapest matching leaf and the cost of deleting the leaf.

function `fetch`(l, t)
   Fetches the posting belonging to label l from the index I_t $(t \in \{struct, text\})$. Returns a new list L that is initialized from the nodes the posting entries refer to.

function `merge`(L_L, L_R, c_{ren})
   Returns a list L consisting of all entries from the distinct lists L_L and L_R. For each entry copied from L_R (but not L_L) the embedding cost is incremented by c_{ren}.

function `join`(L_A, L_D, c_{edge})
   Returns a new list L that consists of copies of all entries from L_A that have descendants in L_D. Let $e_A \in L_A$ be an ancestor and $[e_{D_1}, \ldots, e_{D_m}]$ be the interval in L_D such that each interval entry is a descendant of e_A. The embedding cost of the copy of e_A is set to $\min\{\, distance(e_A, e_{D_i}) + embcost(e_{D_i}) \mid 1 \leq i \leq m \,\} + c_{edge}$.

function `outerjoin`$(L_A, L_D, c_{edge}, c_{del})$
   Returns a new list L that consists of copies of all entries from L_A. Let $e_A \in L_A$ be an entry. If e_A does not have a descendant in L_D then the embedding cost of the copy of e_A is set to $c_{del} + c_{edge}$. Otherwise, let $[e_{D_1}, \ldots, e_{D_m}]$ be the interval in L_D such that each interval entry is a descendant of e_A. The embedding cost of the copy of e_A is set to $\min(c_{del}, \min\{\, distance(e_A, e_{D_i}) + embcost(e_{D_i}) \mid 1 \leq i \leq m \,\}) + c_{edge}$.

function `intersect`(L_L, L_R, c_{edge})
   Returns a new list L. For each pair $e_L \in L_L$, $e_R \in L_R$ such that $pre(e_L) = pre(e_R)$, appends a copy of e_L to L. The embedding cost of the new entry is set to $embcost(e_L) + embcost(e_R) + c_{edge}$.

function `union`(L_L, L_R, c_{edge})
   Returns a new list L that consists of all entries from the lists L_L and L_R. If a node is represented in one list only (say by $e_L \in L_L$) then the embedding cost of the new entry is set to $embcost(e_L) + c_{edge}$. Otherwise, if there are entries $e_L \in L_L$, $e_R \in L_R$ such that $pre(e_L) = pre(e_R)$, then the embedding cost of the new entry is set to $\min(embcost(e_L), embcost(e_R)) + c_{edge}$.

function `sort`(n, L)
   Sorts L by the embedding cost of its entries. Returns the first n entries of L.

6.5   Finding the Best Root-Cost Pairs

Our general algorithm (see Figure 4) for the approximate query-matching problem makes use of the ideas presented in the previous subsections: It takes the expanded representation of an `approXQL` query as input, uses indexes to access the nodes of the data tree, and performs operations on lists to compute the embedding images recursively.

The algorithm expects as input a node u of an expanded query representation, a cost c_{edge} of the edge leading to u, and a list L_A of ancestors. The indexes I_{struct} and I_{text}, used by function `fetch`, are global parameters. Let u be the root of the expanded representation of a query and $[\,]$ be an empty list. Then

$$\texttt{sort}(n, \texttt{primary}(u, 0, [\,]))$$

returns a cost-sorted list of the best n root-cost pairs.

---

function `primary`(u, c_{edge}, L_A)
  case $reptype(u)$ **of**
    leaf: $L_D \leftarrow$ `fetch`$(label(u), type(u))$
         foreach $(l, c_{ren}) \in renamings(u)$ **do**
           $L_T \leftarrow$ `fetch`$(l, type(u))$
           $L_D \leftarrow$ `merge`(L_D, L_T, c_{ren})
         return `outerjoin`$(L_A, L_D, c_{edge}, delcost(u))$
    node: $L_D \leftarrow$ `fetch`$(label(u), type(u))$
         $L_D \leftarrow$ `primary`$(child(u), 0, L_D)$
         foreach $(l, c_{ren}) \in renamings(u)$ **do**
           $L_T \leftarrow$ `fetch`$(l, type(u))$
           $L_T \leftarrow$ `primary`$(child(u), 0, L_T)$
           $L_D \leftarrow$ `merge`(L_D, L_T, c_{ren})
         if u has no parent **then return** L_D
         else return `join`(L_A, L_D, c_{edge})
    and: $L_L \leftarrow$ `primary`$(left\_child(u), 0, L_A)$
         $L_R \leftarrow$ `primary`$(right\_child(u), 0, L_A)$
         return `intersect`(L_L, L_R, c_{edge})
    or: $L_L \leftarrow$ `primary`$(left\_child(u), 0, L_A)$
         $L_R \leftarrow$ `primary`$(right\_child(u), edgecost(u), L_A)$
         return `union`(L_L, L_R, c_{edge})

Fig. 4. Algorithm `primary` finds the images of all approximate embeddings of a query.

The depicted algorithm is simplified. It allows the deletion of *all* query leaves, which is forbidden by Definition 4. To keep at least one leaf, the full version of the algorithm rejects data subtrees that do not contain matches of any query leaf. Furthermore, the full version uses dynamic programming to avoid the duplicate evaluation of query subtrees.

Let s be the maximal number of data nodes that have the same label and let l be the maximal number of repetitions of a label along a path in the data tree. The join functions need $O(s \cdot l)$ time; all other functions need $O(s)$ time. If n is the number of query selectors then the expanded representation has $O(n)$ nodes. The algorithm performs $O(n^2)$ node evaluations, each resulting in at most $O(r)$ function calls, where r is the maximal number of renamings per selector. The overall time complexity of algorithm `primary` is $O(n^2 \cdot r \cdot s \cdot l)$.

7   Schema-Driven Query Evaluation

The main disadvantage of the direct query evaluation is the fact that we must compute *all* approximate results for a query in order to retrieve the *best n*. To find *only* the best n results, we use the *schema* of the data tree to find the best k embedding images, which in turn are used as "second-level" queries to retrieve the results of the query in the data tree. The sorting of the second-level queries guarantees that the results of these queries are sorted by increasing cost as well.

7.1   On the Relationship between a Data Tree and Its Schema

In a data tree constructed from a collection of XML documents, many subtrees have a similar structure. A collection of sound storage media may contain several CDs that all have a title, a composer, or both. Such data regularities can be captured by a *schema*. A schema is similar to a DataGuide [8].

Definition 13 (Label-type path). *A label-type path* $(l_1, t_1).(l_2, t_2) \ldots (l_n, t_n)$ *in a tree is a sequence of label-type pairs belonging to the nodes along a node-edge path that starts at the tree root.*

Definition 14 (Schema). *The schema of a data tree is a tree that contains every label-type path of the data tree exactly once.*

In practice we use *compacted schemata* where sequences of text nodes are merged into a single node and the labels are not stored in the tree but only in the indexes.

Definition 15 (Node class). *A schema node* u *is the class of a data node* v, *denoted by* $u = [v]$, *iff* u *and* v *are reachable by the same label-type path.*

Node v is called an *instance* of u. Every data node v has exactly one class. Node classes preserve the parent-child relationships of their instances. For each triple u, v, w of data nodes holds:

$$v \text{ is a child of } u \Leftrightarrow [v] \text{ is a child of } [u]$$
$$v \text{ and } w \text{ are children of } u \Rightarrow [v] \text{ and } [w] \text{ are children of } [u]$$

Note that the last proposition is an implication: There are node classes that have a common parent in the schema – but no combination of their instances has a common parent in the data tree.

　　We have seen that the mapping between node instances and their class is a function. A node class preserves the labels, the types, and the parent-child relationships of its instances. The same properties hold for embeddings. We can therefore establish a simple relationship between a data tree, its schema, and their included trees (i. e., subgraphs of trees that fulfill the tree properties):

Definition 16 (Tree class). *Let* T *be an included tree of a data tree. The image of an embedding of* T *in the schema is called tree class of* T.

Every included data tree has exactly one tree class, which follows from Definition 14. Embeddings are transitive because all three properties of Definition 1 are transitive. The existence of tree classes and the transitivity of embeddings have an interesting implication: If we have an algorithm that finds the images of all approximate embeddings of a query in the data tree then we can use the same algorithm to find all tree classes of embeddings in the schema of the data tree. We present an adapted version of algorithm `primary` in the following subsection.

　　Not every included schema tree T is a tree class. It is a tree class only if there are "reverse embeddings" from T into the data tree such that each

embedding result contains an instance of T directly. Each result found this way is an approximate result of the query. In Section 7.3, we present an algorithm `secondary` that uses the embedding images found by algorithm `primary` as "second-level" queries to find the results of the original query.

7.2 Finding the Best k Second-Level Queries

The selection of the best k second-level queries using a schema is a straightforward extension of the direct query-evaluation algorithm introduced in Section 6. There, we have tracked the best embedding cost per query subtree and per *data* subtree. Now we track the *images* of the *best k* embeddings (and their costs) per query subtree and per *schema* subtree. We extend the list entries by a value *label* and a set *pointers* yielding the following structure:

$$e = (pre, bound, pathcost, inscost, embcost, label, pointers).$$

The component *label* is initialized by the label of the matching query node; the set *pointers* contains references to descendants of the schema node represented by e. If e represents an embedding root then e and the entries reachable from the pointer set form a second-level query with the embedding cost $embcost(e)$.

   To find not only the best second-level query per schema subtree but the best k ones, we use list *segments*. Recall that lists are sorted by the preorder numbers of the list entries. A segment is a sequence of list entries that have the same preorder number but different embedding costs. All entries of a fixed segment represent embedding images of the *same* query subtree in the *same* schema subtree. Segments are sorted by embedding cost in ascending order. The prefix of k entries of a segment represents the best k embeddings of a certain query subtree in a certain schema subtree. To find the best k second-level queries, only four functions of algorithm `primary` must be adapted:

function `join`(L_A, L_D, c_{edge}, k)

   Returns a new list L. Let $e_A \in L_A$ be an entry. The function calculates the embedding cost of e_A with respect to all entries in all segments in L_D that represent descendants of e_A. For each entry e_D among the k descendants with the lowest costs, a copy of e_A is appended to L. The embedding cost of the copy is set to $distance(e_A, e_D) + embcost(e_D) + c_{edge}$ and its pointer set is initialized with e_D.

function `outerjoin`$(L_A, L_D, c_{edge}, c_{del}, k)$

   Works like the `join` function but additionally calculates the embedding cost $c = c_{edge} + c_{del}$ of each entry $e_A \in L_A$ that does not have a descendant in L_D. If c is among the lowest k embedding costs in the segment generated for e_A then a copy of e_A is appended to L such that the segment for e_A remains sorted. The embedding cost of the new entry is set to c and its pointer set remains empty.

function `intersect`(L_L, L_R, c_{edge}, k)

   Returns a new list L. Let S_L and S_R be segments in L_L and L_R, respectively, that represent the same schema node. The function chooses from both segments the k pairs with the smallest sum of the embedding costs. For each pair (e_L, e_R), a copy of e_L is appended to L. Its embedding cost is set to $embcost(e_L) + embcost(e_R) + c_{edge}$ and its pointer set is initialized with $pointers(e_L) \cup pointers(e_R)$.

function `union`(L_L, L_R, c_{edge}, k)

> Returns a new list L. Let S_L and S_R be segments in L_L and L_R, respectively, that represent the same schema node. The function merges S_L and S_R and copies the prefix of the best k entries to the end of L. If S_L (S_R) does not have a corresponding segment then a copy of S_L (S_R) is appended to L. The embedding cost of each entry appended to L is increased by c_{edge}.

The algorithm `primary` also takes an additional parameter k and passes it to the four modified functions. If u is the root of an expanded query representation and I_{text}, I_{struct} are the indexes of a schema, then

$$\texttt{sort}(k, \texttt{primary}(u, 0, [\,], k))$$

returns the best k second-level queries for the original query.

7.3   Finding the Results of a Second-Level Query

The adapted version of algorithm `primary` returns a list of second-level queries sorted by embedding cost. Using an algorithm `secondary`, each second-level query must be executed against the data tree in order to find all approximate results for the original query.

   As a slight difference to the theoretical setting in Section 7.1, algorithm `primary` does not return embedding images but "skeletons" of embedding images that do not represent the inserted nodes (because the cost of the nodes to insert has been derived from the encoding of the schema). Fortunately, it is not necessary to know the nodes inserted implicitly between two skeleton nodes u and v since all pairs of instances of u and v have by definition the same distance as u and v (see Section 7.1).

   To find all instances of a schema node, we propose *path dependent postings*. A path dependent posting is a sorted list that contains all node instances of a certain schema node, represented as preorder-bound pairs. A *secondary index* I_{sec} maps the nodes of the schema to their postings. A key for I_{sec} is constructed by concatenating the preorder number of a node of a second-level query (which represents a schema node) and the label of the query node: $pre(u)\#label(u)$. Figure 5 shows a simple algorithm that finds all exact embeddings of a second-level query (represented by e_A) in a data tree.

7.4   An Incremental Algorithm for the Best-n-Pairs Problem

So far, we have seen how to find the best k second-level queries and how to find all results for each second-level query. However, we are interested in the *best n results* for a query. Unfortunately, there is no strong correlation between k and n; some second-level queries may retrieve many results, some may not return any result at all. Therefore, a good intial guess of k is cruical and k must be incremented by δ if the first k second-level queries do not retrieve enough results. Fortunately, the increase of k does not invalidate the previous results: The list L_P returned by algorithm `primary` for a certain k is a prefix of the list L'_P returned for a $k' > k$.

function secondary(e_A)	$L_R \leftarrow [\,]$; $k_{prev} \leftarrow 0$
$L_A \leftarrow I_{sec}(\,pre(e_A)\#label(e_A)\,)$	**while** $\|L_R\| < n$ **do**
foreach e_D in $pointers(e_A)$ **do**	$L_P \leftarrow$ sort(k, primary($u, 0, [\,], k$))
$L_D \leftarrow$ secondary(e_D)	erase the first k_{prev} entries from L_P
$L_T \leftarrow [\,]$	$k_{prev} \leftarrow k$; $k \leftarrow k + \delta$
foreach data node u in L_A **do**	**foreach** $e_P \in L_P$ **do**
if u has a descendant in L_D **then**	$L_S \leftarrow$ secondary(e_P)
add u to L_T	**foreach** data node u in L_S **do**
$L_A \leftarrow L_T$	**if** $pre(u)$ is not in L_R **then**
return L_A	add $(pre(u), embcost(e_P))$ to L_R

Fig. 5. The function finds all exact results for a second-level query.

Fig. 6. An incremental algorithm for the best-n-pairs problem.

Our incremental algorithm, depicted in Figure 6, erases at each step the prefix of all second-level queries that have already been evaluated.

Recall from Section 6 that the time complexity of all functions used by algorithm primary is bound by $O(s \cdot l)$, where s is the selectivity and l is recursivity of the data tree. In the following, we use the letters s_s to denote the selectivity in the schema and s_d to denote the maximal number of instances of a node class. The time complexity of the functions adapted in Section 7.2 rises by the factor $k^2 \cdot \log k$, which is the time needed to compute sorted segments of size k. Therefore, the time needed to generate k second-level queries is $O(n^2 \cdot r \cdot s_s \cdot l \cdot k^2 \cdot \log k)$. The evaluation time of k second-level queries is $O(s_d \cdot m)$, where m is the number of nodes of a second-level query.

8   Experiments

In this section, we present selected (but typical) results of the experiments we have carried out to evaluate the efficiency of our algorithms.

8.1   Test Settings

In order to have a high level of control over the characteristics of the data used in our experiments, we employed the XML data generator described in [1]. We varied several parameters of the generator (e.g., the number of elements per document, the total number of terms, and the distribution of terms) and tested our algorithms using the data created for those parameters. Here, we exemplarily present the results of a single test series: We use a document collection that consists of 1,000,000 elements, 100,000 terms, and 10,000,000 term occurrences (words). There are 100 different element names so that on average 10,000 elements share the same name. The words follow a Zipfian frequency distribution.

All queries used in our experiments are produced by a simple generator for approXQL queries. The generator expects a query pattern that determines the structure of the query. A query pattern consists of templates and operators.

The query generator produces approXQL queries by filling in the templates with names and terms randomly selected from the indexes of the data tree. For each produced query, the generator also creates a file that contains the insert costs, the delete costs, and the renamings of the query selectors. The labels used for renamings are selected randomly from the indexes. From the set of tested query patterns we exemplarily choose three patterns that represent a "simple path query", a "small Boolean query", and a "large Boolean query", respectively:

query pattern 1	`name[name[name[term]]]`
query pattern 2	`name[name[term and (term or term)]]`
query pattern 3	`name[name[name[term and term and (term or term)] or` `              name[name[term and term]]] and name]`

For each query pattern and each collection, we created three sets of queries. The sets differ in the number of renamings $(0, 5, 10)$ per query label. Each set contains 10 queries.

All tests have been carried out on a 450 MHz Pentium III with 256 MB of memory, running Linux. Our system is implemented in C++ on top of the Berkeley DB [4].

8.2   Test Results

Our tests results show that the schema-driven query evaluation is faster than the direct evaluation if n, the number of results, is small. For some queries the schema-based algorithm is faster even if all results are requested $(n = \infty)$.
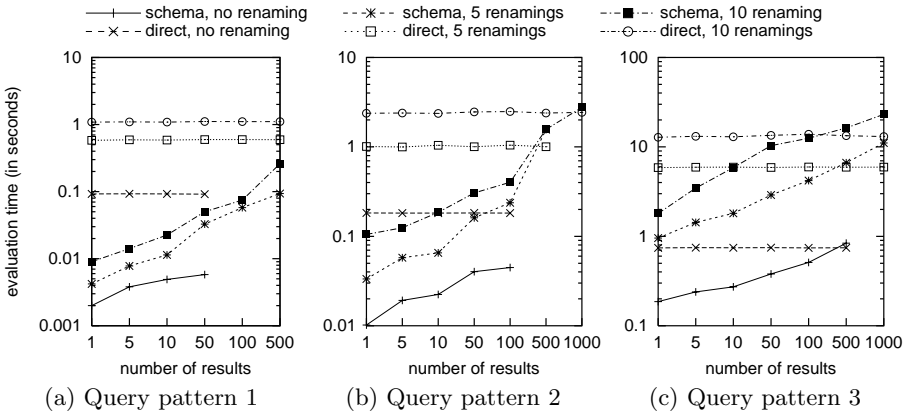

Fig. 7. Evaluation times of the three query patterns.

Each diagram depicted in Figure 7 shows the evaluation time of a query pattern with respect to different numbers of renamings per node and with respect to

the schema-based algorithm (labeled "schema") and the direct algorithm ("direct"). Any point in the diagrams is the mean of the evaluation time of 10 queries randomly generated for the same pattern. Note that the y-axis has logarithmic scale. Figure 7(a) shows the evaluation time of the path query. The schema-based query evaluation outperforms the direct evaluation in all cases – even if no renamings are permitted. This is due to the facts that all second-level queries generated by the algorithm `primary` have at least one embedding in the data tree (each second-level query is a label-type path) and that the postings of the secondary index are much shorter than the postings of I_{struct} and I_{text}. Figure 7(b) displays the evaluation times of the small Boolean query. The diagrams show that the execution time of the schema-based algorithm rises slightly. The reason is that some generated queries may find no results and thus, a larger k must be chosen. However, for small values of n, the schema-based algorithm is always faster than the algorithm for direct evaluation. The larger size of query pattern 3 again increases the average execution times of the algorithms, particularly if 10 renamings per node must be tested (see Figure 7(c)). For small values of n and few renamings, however, the schema-driven algorithm is still fast and outperforms the direct evaluation.

9   Conclusion

In this paper, we introduced an approach to find approximate results for tree-pattern queries using cost-based query transformations. By adjusting the costs of the transformations, our model can be adapted to different types of XML documents. However, the development of domain-specific rules for choosing basic transformation costs is a topic of future research.

We presented and compared two polynomial-time algorithms that retrieve the best n results for a query. We have shown that the schema-driven query-evaluation outperforms the direct evaluation if n is smaller than the total number of results. A further advantage of the schema-based approach is the incremental retrieval: Once the best k second-level queries have been generated, they can be evaluated successively, and the results can be sent immediately to the user.

More details about the schema-driven evaluation of approXQL queries can be found in the extended version [13] of this paper.

References

1. A. Aboulnaga, J.F. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *Proceedings of WebDB'01*, 2001.
2. A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*, Chapter 14: Approximate Tree Pattern Matching. Oxford University Press, 1997.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Longman, 1999.
4. The Berkeley DB. Sleepycat Software Inc., 2000. http://www.sleepycat.com.
5. A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1), 2000.

6. T.T. Chinenyanga and N. Kushmerick. Expressive retrieval from XML documents. In *Proceedings of SIGIR*, 2001.
7. N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *Proceedings of SIGIR*, 2001.
8. R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured data. In *Proceedings of VLDB*, 1997.
9. T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Proceedings of Combinatorial Pattern Matching*, 1994.
10. P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Finland, 1992.
11. J. Robie, J. Lapp, and D. Schach. XML query language (XQL), 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.
12. T. Schlieder. ApproXQL: Design and implementation of an approximate pattern matching language for XML. Report B 01-02, Freie Universität Berlin, 2001.
13. T. Schlieder. Schema-driven evaluation of ApproXQL queries. Report B 02-01, Freie Universität Berlin, 2002.
14. K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979.
15. A. Theobald and G. Weikum. Adding relevance to XML. In *Proceedings of WebDB'00*, 2000.
16. K. Zhang. A new editing based distance between unordered labeled trees. In *Proceedings of Combinatorial Pattern Matching*, 1993.