# Schema-Driven Evaluation of ApproXQL Queries

Torsten Schlieder*

Freie Universität Berlin

schlied@inf.fu-berlin.de

**Abstract**

Query engines for heterogeneous collections of XML data should retrieve exact results – but also answers that are similar to the query. In this paper, we present a simple pattern matching language, 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 present 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 $n$th 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 in detail 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 [BR99]. Unfortunately, the most models (with some recent exceptions, e.g., [TW00, CK01, FG01]) 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 [BC00]. 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.

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

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

```
/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 phrase a query that exactly matches the mentioned cases – 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 allow the renaming of query elements, the insertion of elements into the query, and the 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 the a domain expert.

We present 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 $n$th 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 for small values of $n$, whereas the pruning strategy should by used if (almost) all approximate results are requested.

The paper is organized as follows: In the next section, we introduce the syntax of approXQL queries. Section 3, we show how queries and collections of XML documents can be interpreted as labeled, typed trees. Based on these interpretations, we introduce the semantics of our query language in Section 4. In Section 5, we present the direct query evaluation. The adaption of the algorithm for schema-based query evaluation is the topic of 6. We analyze the results of our experiments in Section 7, review related work in Section 8, and conclude in Section 9.

## 2 The ApproXQL Query Language

ApproXQL is a simple pattern matching language for XML. The syntactical subset of the language that we will use throughout the paper consists of the following constructs:

- name selectors like "cd", "composer", and "title",
- text selectors, which match words and phrases in text sections and attributes,
- the containment operator "`[]`", and
- the logical connectors "`and`" and "`or`".

The following query selects CDs containing piano concertos composed by Rachmaninov:

```
cd[title["piano" and "concerto"] and composer["rachmaninov"]]        (Q1)
```

Note that the user does not need to know how the year 2001 is modeled in the original documents because the text selectors match both cases.

The operator "`or`" can be used to specify alternative paths through the XML documents:

```
cd[title["piano" and ("concerto" or "sonata")] and
  (composer["rachmaninov"] or performer["ashkenazy"])]        (Q2)
```

A more detailed description of approXQL and particularly its syntactical constructs to express user preferences can be found in [Sch01].

# 3 Trees: Modeling Queries and Documents

The semantics of an approXQL query is based on the similarity between trees. In order to introduce this semantics, we first map queries and documents to trees: Queries are broken up into conjunctive trees. All documents of a collection are mapped to a single data tree.

## 3.1 Trees

A *rooted tree* is a structure $T = (N, E, r)$, where $N$ is a finite set of *nodes*, $E \subseteq N \times N$ is a finite set of *edges*, and $r \in N$ is a node that forms the *root* of $T$. A *path* in a tree is a sequence of edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k)$. The path starts from node $u_1$, ends at node $u_k$, and has the *length* $k - 1$. A node $v$ is a *descendant* of a node $u$ if there is a path between $u$ and $v$. In this case, we call $u$ an *ancestor* of $v$. An *included tree* $T' = (N', E', u)$ in $T$ is a tree, where $N' \subseteq N$ and $E' = E \cap (N' \times N')$. A *subtree* of $T$ is a tree rooted at a node $u \in T$ that consists of all nodes and edges of $T$ along the paths from $u$ to the leaves of $T$. A rooted tree in which for each node $u$ attributes $label(u)$ and $type(u)$ are defined is called *labeled, typed tree*.

## 3.2 Tree Interpretation of Queries

Conjunctive approXQL queries can be interpreted as labeled, typed trees: 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 query $Q_1$.

If a query contains "`or`" operators then we create its disjunctive normal form (DNF). The DNF of is obtained by a top-down decomposition of the query. At each hierarchy level the DNF of the (flat) Boolean expression is created, treating the included hierarchy operators as atomic expressions. All selectors that are ancestors of the root selector of the actual hierarchy level are replicated for each created conjunct. Query $Q_2$ consists of two "`or`"-operators and thus can be converted into $2^2 = 4$ conjunctive queries:

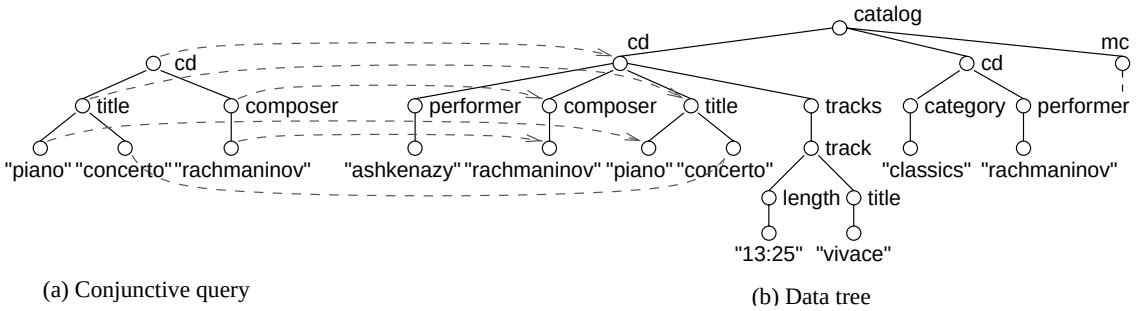
```
cd[title["piano" and "concerto"] and composer["rachmaninov"]] or
cd[title["piano" and "concerto"] and performer["ashkenazy"]] or
cd[title["piano" and "sonata"] and composer["rachmaninov"]] or
cd[title["piano" and "sonata"] and performer["ashkenazy"]].
```

The *separated representation* $\mathbb{Q}$ of a query is a set of labeled, typed trees $\{T_{Q_1}, T_{Q_2}, \ldots, T_{Q_4}\}$, where each tree $T_{Q_i}$ represents a conjunct of the DNF of the query.

## 3.3 Tree Interpretation of XML Documents

We model XML documents as labeled trees consisting of two node types: *text* nodes represent text data as well as attribute values; nodes of type *struct* represent elements and attributes. The name of an element is used as node label. In order to enable a matching on the level of terms (which is typical for information-retrieval systems), we decompose all text sequences into words. 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 edges between this node and the roots of the document trees. The resulting tree is called *data tree*. Figure 1(b) shows an example of a data tree.




(a) Conjunctive query

(b) Data tree

Figure 1: Embedding of a query tree in a data tree.

# 4 Querying by Approximate Tree Embedding

In this section, we introduce the semantics of evaluating an approXQL query. We first define an embedding function that maps a query tree 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.

## 4.1 The Tree-Embedding Formalism

Our definition of tree embedding is inspired by the *unordered path inclusion problem* proposed by Kilpeläinen [Kil92]. Unordered path inclusion is defined as an injective function that preserves labels and parent-child relationships, but not the order of the siblings. We discard the injectivity property of the path inclusion problem in order to get a function that is efficiently computable[1]

**Definition 4.1 (Embedding)** *Let $T_Q = (N_Q, E_Q, r_Q)$ be a query tree and $T_D = (N_D, E_D, r_D)$ be a data tree. A mapping $f : N_Q \to N_D$ is called an embedding of $T_Q$ into $T_D$ if and only if for all $u_Q, v_Q \in N_Q$ holds:*

1. $u_Q = v_Q \Rightarrow f(u_Q) = f(v_Q)$ *(f is a function),*
2. $type(u_Q) = type(f(u_Q))$ *(f is type preserving),*
3. $label(u_Q) = label(f(u_Q))$ *(f is label preserving),*
4. $(u_Q, v_Q) \in E_Q \Leftrightarrow (f(u_Q), f(v_Q)) \in E_D$ *(f is parent-child preserving).*

Let $u_Q$ be a node of a query tree. We call the data node $f(u_Q)$ a *match* of $u_Q$. 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 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 query tree 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.

## 4.2 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 query tree by inserting a node, deleting a node, or renaming the label of a node. In contrast to the tree edit distance [Tai79], 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 an

[1] The injectivity of the embedding function together with the implicit relaxation of the parent-child-relationship to an ancestor-descendant relationship (Section 4.2) would lead to an instance of the *unordered tree inclusion problem*, which is NP-complete [Kil92].

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. Another motivation for the restrictions is the efficient computability of the results of *all* transformed queries. For example, it is not allowed to delete or rename a node previously inserted in the query.

**Definition 4.2 (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 expects the matches of a subtree 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 4.3 (Deletion of inner nodes)** *A deletion removes an inner node $u_Q$ (except the root) together with its incoming edge and connects the outgoing edges of $u_Q$ with the parent of $u_Q$.*

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 4.4 (Deletion of leaves)** *A deletion removes a leaf $u_Q$ together with its incoming edge if and only if the parent of $u_Q$ has two or more children (including $u_Q$) that are leaves of the query.*

The deletion of leaves adopts the concept of "coordination level match" [BR99], which is a simple querying model that establishes ranking for queries of "and"-connected search terms. Documents containing all $n$ terms of the query get the highest score, documents containing $n-1$ terms get the second-highest score, and so on.

**Definition 4.5 (Renaming)** *A renaming changes the label of a node.*

A renaming of a node $u_Q$ changes the search space of the query subtree rooted at $u_Q$. For example, the renaming of the query root from cd to mc obviously shifts the search space from CDs to MCs; the renaming of composer to performer changes the context in which the term "rachmaninov" is expected; and the renaming of the term "concerto" to "sonata" changes the context of the text selector. Note, that a node typically has only few semantically meaningful renamings.

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

**Definition 4.6 (Cost)** *The cost of a basic transformation is a non-negative number.*

There are several possibilities to assign costs to transformations (see [Sch01]). Throughout this paper, we choose the simplest variant: We bind to cost to the label of the inserted node, the label of the deleted node, and the pair of old and new label of a renamed node, respectively.

## 4.3 The Approximate Query-Matching Problem

In this section, we formally define the approximate query-matching problem. We first define transformation sequences and the embedding cost of a transformed query tree as the sum of the costs of all applied basic transformations.

**Definition 4.7 (Transformation sequence)** *Let $T_Q$ be a query tree. A transformation sequence $t_1, t_2, \ldots, t_n$ is a series of basic query transformations applied to $T_Q$ such that all deletions precede all renamings and all renamings precede all insertions.*

The order of transformation sequences has an intuitive semantics: All destructive operations (deletions) precede all global restructurings (permutations), all global restructurings precede all local restructurings (value changes), and all local restructurings precede all constructive operations (insertions). The order of transformations is one of the key concepts that allows an implementation with a favourable time complexity (see Section 5).

**Definition 4.8 (Embedding cost)** *Let $T_Q$ be a query tree and $T'_Q$ be a tree derived from $T_Q$ by a transformation sequence $t_1, t_2, \ldots, t_n$. The embedding cost of $T'_Q$ is defined as*

$$embcost(T'_Q) = \sum_{i=1}^{n} cost(t_i).$$

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

**Definition 4.9 (Query closure)** *The closure $\mathbb{Q}^*$ of a query $Q$ is the set of all transformed queries that can be derived from the separated representation of $Q$ using transformation sequences.*

The closure of an approXQL query $Q$ is a set of query trees. To find all approximate results for $Q$, we execute each query tree $T_Q \in \mathbb{Q}^*$ against the data tree $T_D$. Executing a query means finding a (possibly empty) set of embeddings of $T_Q$ in $T_D$ according to Definition 4.1. All embeddings that have the same root are collected in an embedding group:

**Definition 4.10 (Embedding group)** *An embedding group is a set of pairs, where each pair $(f, c)$ consists of an embedding $f$ and its cost $c$. 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 smallest embedding cost:

**Definition 4.11 (Approximate query-matching problem)** *Given a data tree and the closure of a query, locate all embedding groups. Represent each group by a pair $(u_Q, c)$, where $u_Q$ is the common root of the embeddings in the group and $c$ is the smallest cost of all embeddings in the group.*

We call the pair $(u_Q, c)$ 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:

**Definition 4.12 (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.

# 5 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 will see that the explicit creation of transformed queries is not necessary. Moreover, we will see that the images of all approximate embeddings of a query can be found in polynomial time with respect to the number of nodes in the data tree or the schema, respectively. 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 represent all possible insertions of query nodes by 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.

The algorithm that we introduce in this section is in fact more general: It can be utilized to find the images of the best $k$ embeddings of a query in a schema (see Section 6). The images of schema embeddings can be used as "second-level" queries. 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.

## 5.1 The Expanded Representation of a Query

Many transformed queries in the closure of an approXQL query are quite similar; they often differ only in some inserted nodes. We call a query tree that is derived from a query tree using a sequence of deletions and renamings (but no insertions) a *semi-transformed query*. In Section 5.2, we shall see how the insertions can be determined from the data tree. The *expanded representation* of a query $Q$ encodes all distinct semi-transformed queries that can be derived from the separated representation of $Q$. We introduce the expanded representation of a query by describing its four *representation types*:

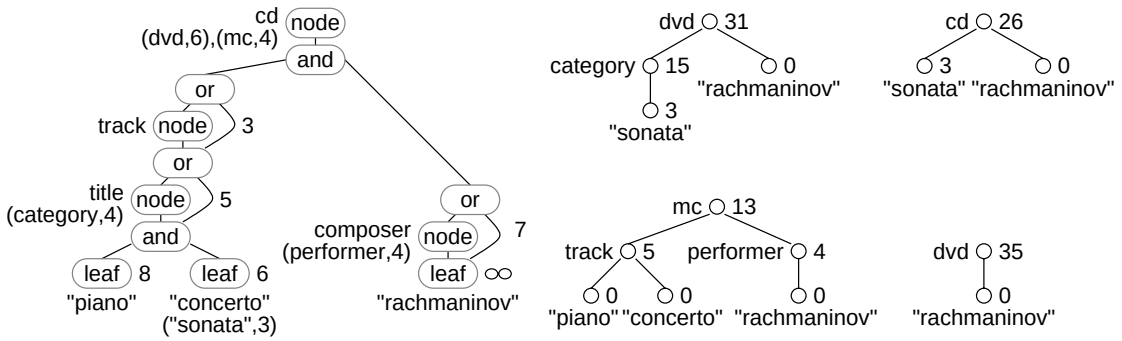
leaf : A node of representation type leaf represents all leaves in all semi-transformed queries that are derived from the same leaf in the original query. Consider the expanded query representation shown in Figure 2(a): The middle leaf represents the "concerto" node of the original query. It is labeled 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.

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. The top-level node in Figure 2(a) represents the cd node of the original query and its renamings dvd and mc that have the costs 6 and 4, respectively.

and : Each 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 may be deleted and has therefore an or-parent.

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*




(a) Expanded representation

(b) Four out of 84 semi-transformed queries

Figure 2: Expanded representation and semi-transformed queries derived from the query `cd[track[title["piano" and "concerto"]] and composer["rachmaninov"]]`.

cost of approximate embeddings of the subtree rooted at the node. Node insertions in the subtree may increase the costs. A semi-transformed query can be derived from the expanded representation by following a combination of paths from the root to the leaves. For example, the upper-left query in Figure 2(b) is constructed as follows: Start at the top-level node. Choose the renaming dvd and initialize the cost $c$ with value 6. Follow the left child of the top-level and node and choose the right child of the or node. Add the delete cost 3 of node track to $c$. Follow the left child of the lower-left or node and add the rename cost 4 of category to $c$. Proceed to the left child of the and node and add the delete cost 8 to $c$. Then take the right child of and node and add the rename cost 3 belonging to "sonata" to $c$. Continue with the right child of the top-level and node and follow the right child of the or node. Add the delete cost 7 to $c$ yielding an embedding cost 31 for the entire semi-transformed query.

For each node $u_Q$ of the expanded query representation, we define a number of attributes, which are used by our query-evaluation algorithm:

| attribute | value | defined for |
|---|---|---|
| $reptype(u_Q)$ | representation type of $u_Q$ (and, or, node, leaf) | all nodes |
| $type(u_Q)$ | type of $u_Q$ (*struct*, *text*) | node, leaf |
| $label(u_Q)$ | label of $u_Q$ | node, leaf |
| $renamings(u_Q)$ | set of label-cost pairs for $u_Q$ | node, leaf |
| $delcost(u_Q)$ | cost of deleting $u_Q$ | leaf |
| $edgecost(u_Q)$ | cost assigned to the edge leading to the right child of $u_Q$ | or |

## 5.2 Encoding of the Data Tree

The embedding of a (transformed) query tree 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 4.1) from "parent-child preserving" to "ancestor-descendant preserving" and define the distance between two nodes $u_D$ and $v_D$ as the sum of the insert costs of all nodes along the path from $u_D$ to $v_D$ (excluding $u_D$ and $v_D$). This modification allows an efficient realization of our algorithm but does not affect any proposition of Section 4.

To test if two nodes are ancestor and descendant of each other and to calculate the distance

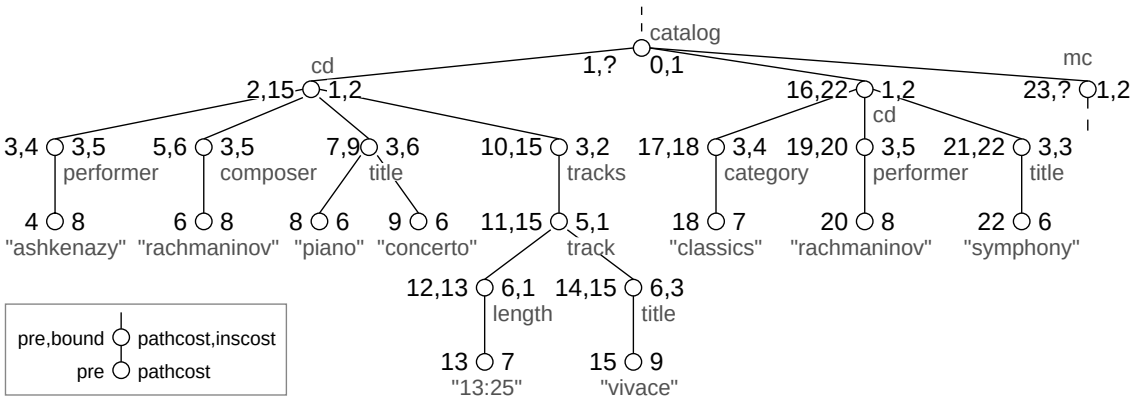



Figure 3: An encoded data tree.

between them, we define the following attributes for the nodes of the data tree:

| attribute | value | defined for |
|---|---|---|
| $pre(u_D)$ | preorder number of $u_D$ | all nodes |
| $bound(u_D)$ | number of the rightmost leaf of the subtree rooted at $u_D$ | inner nodes |
| $pathcost(u_D)$ | sum of the insert costs of all ancestors of $u_D$ | all nodes |
| $inscost(u_D)$ | cost of inserting $u_D$ into a query | inner nodes |

The preorder numbers are assigned during a depth-first traversal of the data tree. Given two data nodes $u_D$ and $v_D$ we can test if $u_D$ is an ancestor of $v_D$ by ensuring the invariant

$$pre(u_D) < pre(v_D) \wedge bound(u_D) \geq pre(v_D).$$

If $u_D$ is an ancestor of $v_D$ then the distance between $u_D$ and $v_D$ is

$$distance(u_D, v_D) = pathcost(v_D) - pathcost(u_D) - inscost(u_D).$$

An example of an encoded schema is shown in Figure 3. 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.

| | |
|---|---|
| 13:25 : 13 | catalog : 1 |
| ashkenazy : 4 | category : 17 |
| classics : 18 | cd : 2,16 |
| concerto : 9 | ... : ... |
| piano : 8 | performer : 3,19 |
| rachmaninov : 6,20 | title : 7,14,21 |
| symphony : 22 | track : 11 |
| vivace : 15 | tracks : 10 |
| (a) Index $I_{text}$ | (b) Index $I_{struct}$ |

Figure 4: The indexes of the data tree shown in Figure 3

A structural index $I_{struct}$ and a text index $I_{text}$ provide access to the nodes of the data tree. The indexes map each label occurring in the data tree to a posting consisting of references to

all nodes that carry the label. Each posting is sorted by the preorder numbers of the nodes the posting entries refer to. The Figures 4(a) and 4(b) show the indexes of the encoded data tree depicted in Figure 3.

## 5.3 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 corresponding node $u_D$. If $u_D$ is a *text* node then *bound* and *inscost* are set to zero. The value *embcost* stores the cost of embedding a query subtree into the data subtree rooted at $u_D$. The value is zero if $u_D$ is the match of a query leaf.

Because a list entry has the same attributes as the represented data node, it can be treated as a node in many respects. For example, we can test the ancestor-descendant relationship between two entries and can compute the distance between them.

Lists are sorted by the preorder numbers of their entries in ascending order. The sort order preserves the topological relationships of the nodes. If a node $u_D$ has descendants $\{v_{D_1}, v_{D_2}, \ldots, v_{D_m}\}$ with label $l$, and the list $L$ represents all nodes with label $l$, then the descendants reside in a closed interval $[e_{D_1}, e_{D_2}, \ldots, e_{D_m}]$ of $L$. For convenience, we use the set notation $e \in L$ to refer to a particular entry of $L$.

## 5.4 Operations on Lists

We now introduce the basic list operations used by our query-evaluation algorithm. 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 $e_{D_1}, e_{D_2}, \ldots, e_{D_m}$, 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 text index $I_{text}$ (if $t = text$) or the structural index $I_{struct}$. 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 $[e_{D_1}, e_{D_2}, \ldots, e_{D_m}]$ 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 $[e_{D_1}, e_{D_2}, \ldots, e_{D_m}]$ 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 $\min(c_{del}, \min\{\, \mathit{distance}(e_A, e_{D_i}) + \mathit{embcost}(e_{D_i}) \mid 1 \leq i \leq m \,\}) + c_{edge}$.

**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 $\mathit{pre}(e_L) = \mathit{pre}(e_R)$, appends a copy of $e_L$ to $L$. The embedding cost of the new entry is set to $\mathit{embcost}(e_L) + \mathit{embcost}(e_R) + c_{edge}$.

**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 $\mathit{embcost}(e_L) + c_{edge}$. Otherwise, if there are entries $e_L \in L_L$, $e_R \in L_R$ such that $\mathit{pre}(e_L) = \mathit{pre}(e_R)$, then the embedding cost of the new entry is set to $\min(\mathit{embcost}(e_L), \mathit{embcost}(e_R)) + c_{edge}$.

**function** `sort`$(n, L)$

Sorts $L$ by the embedding cost of its entries. Returns the first $n$ entries of $L$.

The paper [Sch01] provides a more detailed view on the realizations of the functions.

## 5.5 Finding the Best Root-Cost Pairs

Our algorithm 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.

Algorithm 1 shows the function `primary`, which implements the query-evaluation algorithm. It expects a node $u_Q$ of an expanded query representation, a cost $c_{edge}$ of the edge leading to $u_Q$, and a list $L_A$ of ancestors as input. We assume that the indexes $I_{struct}$ and $I_{text}$, used by function `fetch`, are global parameters. Let $u_Q$ be the root of the expanded representation of a query and [ ] be an empty list. Then

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

returns a cost-sorted list of the best $n$ root-cost pairs, each representing the root of a result (i.e., a data subtree) together with the embedding cost of the query tree with the lowest embedding cost of all query trees embeddable in this document.

The evaluation of a query consists of alternating top-down and bottom-up sequences. During top-down sequences the Boolean nodes of the expanded query representation are ignored. For any other node $u_Q$, the list $L$ of all data nodes that have the same label and type as $u_Q$ is fetched. If $u_Q$ is a leaf, then the lists belonging to all renamings of $u_Q$ are merged with $L$. If $u_Q$ is an inner node, then $L$ is passed as list of ancestors to any child of $u_Q$. The same happens for all lists belonging to renamings of $u_Q$. The lists resulting from recursive calls to `primary` are merged. Then, a bottom-up sequence starts with a call to the functions `join` or `semijoin`, respectively. At this point, $L_D$ contains the embedding roots of all transformed queries that share the same node $u_Q$ or one of its renamings. $L_D$ is joined with the list fetched for the lowest ancestor of $u_Q$ that has the representation type `node`. The result list of the join is passed to the parent $v_Q$ of $u_Q$. If $v_Q$ is a Boolean node then the list passed from $u_Q$ is combined with the list of another subtree via a `union` or `intersect` operator, and the result is passed to the parent of $v_Q$.

Note, that Algorithm 1 is simplified. First, it allows the deletion of *all* query leaves – which is forbidden by Definition 4.4. To keep at least one leaf, the correct version of the algorithm rejects data subtrees that do not contain matches of any query leaf. Second, the algorithm performs many redundant steps. If, for example, the expanded query depicted in Figure 2(a) is evaluated, then

**Algorithm 1** finds the images of all approximate embeddings of a query.

```
function primary(u_Q, c_edge, L_A)
  case reptype(u_Q) of
    leaf:  L_D ← fetch(label(u_Q), type(u_Q))
           foreach (l, c_ren) ∈ renamings(u_Q) do
             L_T ← fetch(l, type(u_Q))
             L_D ← merge(L_D, L_T, c_ren)
           return outerjoin(L_A, L_D, c_edge, delcost(u_Q))
    node:  L_D ← fetch(label(u_Q), type(u_Q))
           L_D ← primary(child(u_Q), 0, L_D)
           foreach (l, c_ren) ∈ renamings(u_Q) do
             L_T ← fetch(l, type(u_Q))
             L_T ← primary(child(u_Q), 0, L_T)
             L_D ← merge(L_D, L_T, c_ren)
           if u_Q has no parent then
             return L_D
           return join(L_A, L_D, c_edge)
    and:   L_L ← primary(left_child(u_Q), 0, L_A)
           L_R ← primary(right_child(u_Q), 0, L_A)
           return intersect(L_L, L_R, c_edge)
    or:    L_L ← primary(left_child(u_Q), 0, L_A)
           L_R ← primary(right_child(u_Q), edgecost(u_Q), L_A)
           return union(L_L, L_R, c_edge)
```

algorithm `primary` would fetch and merge the lists belonging to the labels "concerto" and "sonata" four times because there exist four different paths to this leaf. Similarly, the query subtree rooted at the title node would be evaluated two times. The full version of the algorithm uses *dynamic programming*. It stores intermediate results in an array and stops a top-down sequence if the results for a particular node are found in the array. The full version performs at most $O(n^2)$ node evaluations, where $n$ is the number of nodes in the original query.

## 5.6 Incremental Retrieval

Algorithm `primary` evaluates all data subtrees containing approximate query matches independent of the cost of intermediate results. The best $n$ root-cost pairs are selected after the evaluation has finished. We propose *pruning* as a means to discard intermediate results whose costs are higher than a fixed threshold.[2].

To integrate threshold-based pruning in our framework, we only have to modify the functions used by algorithm `primary` but not the algorithm itself. In the definition of each function, we change the creation of the result list $L$: A copy of an entry is appended to $L$ only if its embedding cost is less than the threshold $\theta$. Consider, for example, the function `join` on page 10: If the cost $\min\{\, distance(e_A, e_{D_i}) + embcost(e_{D_i}) \mid 1 \le i \le m \,\} + c_{edge}$. calculated for entry $e_A$ is larger than $\theta$ then we do not append $e_A$ to $L$.

If we had a domain of application where we knew in advance which embedding cost $c_{emb}$ belonged to the $n$th result then we could efficiently solve the best-$n$-documents problem with a single-pass algorithm by setting $\theta \leftarrow c_{emb}$. Unfortunately, in the most cases we do have this knowledge, and therefore, we must guess the embedding cost of the $n$th result. If the first pass yields less than $n$ results then we must increase $\theta$. Algorithm 2 sketches this approach.

Algorithm `primary` takes the additional parameter $\theta$ and passes it to its functions, which

[2]We present another strategy for complexity reduction in Section 6, where we use the schema of the data tree to estimate the best $k$ transformed query trees, which are in turn executed against the data tree. The schema-based evaluation and the threshold-based pruning can be combined.

**Algorithm 2** finds all matching document for a query.

**input**: $Q$ – a query.
**output**: the best $n$ root-cost pairs in sorted order.

```
1:  guess an inital value for θ
2:  let u_Q be the root of the expanded representation of Q
3:  let L be an empty list
4:  i ← 0
5:  while |L| < n do
6:      L ← sort(k, primary(u_Q, 0, [ ], θ))
7:      prune the first i entries of L
8:      output the root-cost pairs of L
9:      increment θ
10:     i ← |L|
```

perform the pruning according to $\theta$. At the end of each pass the results found so far can be retrieved to the user. However, at the next pass the prefix of all results found so far must be computed again – it is not possible to provide a threshold for the smallest embedding cost of interest as we will now explain: Assume that we had a threshold $\theta'$ for the lower bound of the embedding costs. If we pruned away an intermediate result whose threshold is below $\theta'$ then we would possibly discard a constituent of a result with a cost larger than the upper bound $\theta$. The only implication we can do is that the embedding cost of the final result is larger than the cost of any of its constituents.

### 5.7 Complexity Analysis

All functions used by algorithm `primary` have a polynomial time complexity with respect to the number of nodes in the data tree. 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 time complexity of the functions `join` and `semijoin` is bound by $O(s \cdot l)$; all other functions have the time complexity $O(s)$. 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

$$O(n^2 \cdot r \cdot s \cdot l).$$

This formula does not include the time to access the index $I_{struct}$ or $I_{text}$, respectively.

## 6 Schema-Based 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. Unfortunately, the number of second-level queries may be exponential with respect to the number of nodes of the original query. In Section 6.3, we therefore propose an adapted version of algorithm `primary` that finds only the *best* $k$ second-level queries. Then, we introduce a simple algorithm `secondary`, which finds all results for a given second-level query. Finally, we present an incremental query-evaluation algorithm that retrieves the best $n$ approximate results for a query given an initial guess for $k$.

## 6.1 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 [GW97]. In this section, we investigate how the schema of a data tree can be used to find all results matching the query approximately. We show that the images of embeddings of query trees in the schema can serve as "second-level-queries" to find the results of the query in the data tree. We also show that, given correct and complete algorithms $\mathcal{A}$ and $\mathcal{B}$, the schema-driven query evaluation finds all results – and only correct results – of an approXQL query with respect to a data tree. Throughout this subsection we use the following abbreviations:

*approximate embedding:* An embedding of a transformed query tree, which is an element of the closure of a query.

*approximate result:* The root of a data subtree that contains directly the image of an approximate embedding. Direct containment requires that the root of the data subtree and the root of the embedding image are identical.

### 6.1.1 Schemata and Node Classes

We now formally define a schema as representation of all distinct label-type paths through the data tree. Then, we introduce the concept of a *node class*, which is a schema node that represents all data nodes reachable by the same label-type path.

**Definition 6.1 (Label-type path)** *A label-type path $(l_1, t_1).(l_2, t_2)\dots(l_n, t_n)$ 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 6.2 (Schema)** *The schema $T_S$ of a data tree $T_D$ is a tree that includes every label-type path of $T_D$ exactly once.*

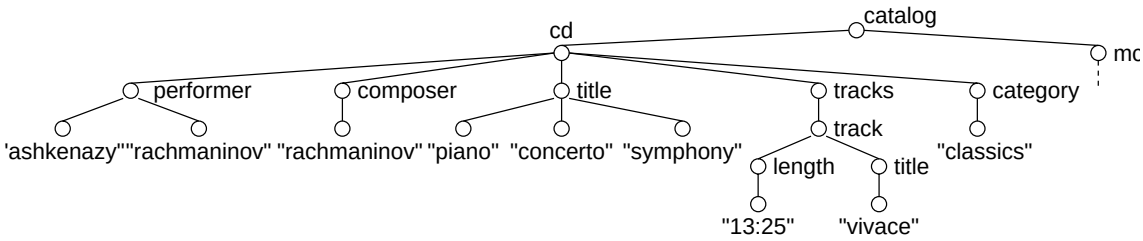



Figure 5: The schema of the data tree depicted in Figure 3.

Figure 5 shows the schema constructed from the data tree depicted Figure 3. The number of nodes of this schema is not significantly smaller than the node number in the corresponding data tree because all distinct leaf labels are represented. The definition of a schema used in this section is a simplification that helps to explain our approach. In the next section, we introduce *compacted schemata*, which have identical properties with respect to the approximate query-matching problem. In compacted schemata, all leaves that have the same parent are merged and the labels of the nodes are stored *only* in index structures. Therefore, an compacted schema is much smaller than the data tree from which it is constructed.

In the following, we assume an arbitrary but fixed approXQL query $Q$ and an arbitrary but fixed data tree $T_D = (N_D, E_D, r_D, type_D, label_D)$. $T_D$ has a schema $T_S = (N_S, E_S, r_S, type_S, label_S)$ according to Definition 6.2.

**Definition 6.3 (Node class)** *A node $v_S \in N_S$ is the class of a node $v_D \in N_D$, denoted by $v_S = [v_D]$, if and only if $v_S$ and $v_D$ are reachable by the same label-type path.*

Node $v_D$ is called an *instance* of $v_S$. Every data node $v_D \in N_D$ has exactly one node class because there is only one label-type path from the root of the data tree to $v_D$. Node classes preserve the parent-child relationships, the ancestor-descendant relationships, and even the common-ancestor relationships of their instances. For each triple $u_D, v_D, w_D \in N_D$ of data nodes holds:

$$\begin{aligned} v_D \text{ is a child of } u_D &\Rightarrow [v_D] \text{ is a child of } [u_D] \\ v_D \text{ is a descendant of } u_D &\Rightarrow [v_D] \text{ is a descendant of } [u_D] \\ v_D \text{ and } w_D \text{ are children of } u_D &\Rightarrow [v_D] \text{ and } [w_D] \text{ are children of } [u_D] \\ v_D \text{ and } w_D \text{ are descendants of } u_D &\Rightarrow [v_D] \text{ and } [w_D] \text{ are descendants of } [u_D] \end{aligned}$$

Note that all propositions are implications: There may be node classes that stand in an ancestor-descendant relationship but *not every* pair of their instances does. Furthermore, there are node classes that have a common ancestor in the schema – but it is possible that *no* combination of their instances has a common ancestor in the data tree. As an example consider the nodes track and category and its common ancestor cd in Figure 5. In the data tree depicted in Figure 3, there is no CD that has both a track and a category.

The example shows that sometimes a pair of sibling nodes of the schema has no pair of instances that are siblings. However, it always holds that two schema nodes that stand in an ancestor-descendant relationship have at least one pair of instances that stand in an ancestor-descendant relationship. We formalize this observation. Let $u_S, v_S \in N_S$ be schema nodes. Then holds

$$\begin{aligned} v_S \text{ is a child of } u_S &\Leftrightarrow \exists u_D, v_D \in N_D \text{ such that } [u_D] = u_S \text{ and } [v_D] = v_S \\ &\quad \text{and } v_D \text{ is a child of } u_D \\ v_S \text{ is a descendant of } u_S &\Leftrightarrow \exists u_D, v_D \in N_D \text{ such that } [u_D] = u_S \text{ and } [v_D] = v_S \\ &\quad \text{and } v_D \text{ is a descendant of } u_D \end{aligned}$$

### 6.1.2 Answering Queries Using a Schema

In this subsection, we show that there are two ways of answering an approXQL query: The direct query evaluation, described in Section 5, and a two-level query evaluation that uses a schema as intermediate step. We prove that both ways yield the same set of document roots.

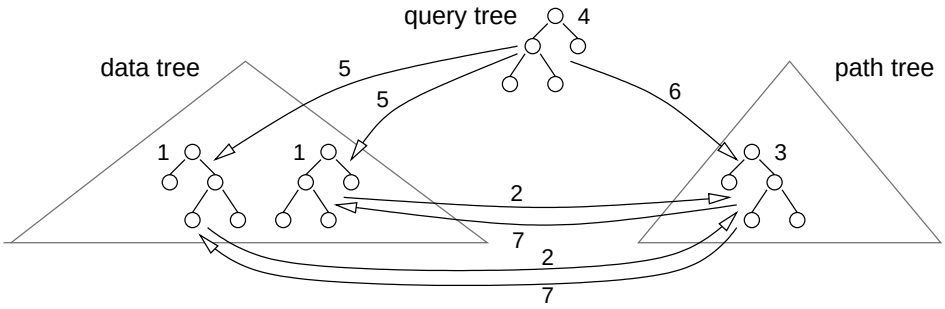



Figure 6: The relationships between a data tree and its schema.

Before we go into detail, we give an overview of our proofs. Figure 6 sketches the main steps. First, we show that every included data tree (1) has an embedding (2) in the schema. The image (3) of the embedding is called *tree class*. Several included data trees may have the same tree class. Then, we show that embeddings are transitive. Given a query tree (4) and exact embeddings in the data tree (5) we can also construct an embedding of the query tree in the schema (6) so that the embedding image (3) is the tree class of (1). We can extend this observation to the closure of a query: If we have an algorithm $\mathcal{A}$ that finds, for all query trees in the closure, the images of all embeddings into a data tree then we can use the same algorithm to find the tree classes of these

images. (But $\mathcal{A}$ may also find embedding images in the schema that are not tree classes.) We then show that every tree class has reverse embeddings (7) that help to locate the instances of the tree class, and thus, to locate all results represented by the tree class. We assume an algorithm $\mathcal{B}$ that finds the roots of all data subtrees (results) containing instances of a tree class. Then $\mathcal{B}(\mathcal{A}(Q))$ finds *all* approximate results of $Q$ (completeness). Using the transitivity of embeddings, we show that $\mathcal{B}(\mathcal{A}(Q))$ finds *only* approximate results of $Q$ (correctness).

**Definition 6.4 (Tree class)** *Let $T'_D$ be an included tree of $T_D$. The tree class of $T'_D$ is the image of an embedding of $T'_D$ in $T_S$.*

Every included tree of the data tree has a tree class, which we will prove in the sequel. Intuitively, the construction of an embedding from any included tree of the data tree into the schema is possible because the schema preserves all vertical relationships of the data tree and also all horizontal relationships required by embeddings. Of course not all horizontal relationships of the data tree are preserved; in particular, the order of siblings is lost and siblings with identical types and labels are mapped to a single node of the schema. But these are exactly the relationships that are not required by embeddings – by definition, an embedding is unordered and not injective. We cannot ensure the opposite direction: Not every included tree of the schema is a tree class because the schema comprises sibling relationships that have no counterparts in the data tree. Consider the schema depicted in Figure 5. The included tree consisting of the cd node and the composer and category branches is not a tree class because there is no CD in our catalog that has both a composer and a category.

**Lemma 6.1 (Tree-class existence)** *Every included tree $T'_D$ of $T_D$ has a tree class in $T_S$.*

**Proof:** We prove the lemma by constructing an embedding $f$ from $T'_D$ in $T_S$ according to Definition 4.1 on page 4. We set $f(u_D) = [u_D]$ for each $u_D$ in the node set of $T'_D$. The node-class mapping $[u_D]$ is well-defined and is a type- and label-preserving function (Definition 6.3). Therefore, $f$ fulfills the first three properties of Definition 4.1. Let $v_D$ be a child of $u_D$. Definition 6.2 states that the children of a data node $u_D$ are mapped to children of the schema node that belongs to $u_D$. That is, $[v_D]$ is a child of $[u_D]$ and therefore $f(v_D)$ is a child of $f(u_D)$. Thus, Property 4 of Definition 4.1 holds and $f$ is an embedding. □

So far we have shown that any included tree of the data tree has a tree class. An implication of this proposition is: Every image of an embedding of a query tree $T_Q$ in the data tree has a tree class. Now, we need a way to find the tree class of the embedding image of $T_Q$ without touching the data tree. Once we have this class, we can use it to find the embeddings of $T_Q$ in the data tree. The following lemma helps to prove that we can in fact find a tree class using the schema only.

**Lemma 6.2 (Embedding transitivity)** *Let $f$ be an embedding of a tree $T_A$ in a tree $T_B$ such that $T'_B$ is the image of $f$. Let $f'$ be an embedding of $T'_B$ in a tree $T_C$ such that $T'_C$ is the image of $f'$. Then there exists an embedding $f''$ from $T_A$ in $T_C$ such that $T'_C$ is the image of $f''$.*

**Proof:** All four properties of Definition 4.1 are transitive. We can therefore construct an embedding $f''$ by defining $f''(v_A) = f'(f(v_A))$ for each $v_A$ in the node set of $T_A$. □

Lemma 6.2 allows us to find the tree classes belonging to the images of all data-tree embeddings of a query tree $T_Q$ without touching the data tree: Whenever $T_Q$ has an embedding in the data tree then the image of the embedding has a tree class $T'_S$. Because embeddings are transitive there is also an embedding from $T_Q$ into the schema such that $T'_S$ is the image. Thus, if we know the images of all schema embeddings of $T_Q$ then those images include all tree classes belonging to data-tree embeddings of $T_Q$. We can easily extend this principle to all query trees in the closure $\mathbb{Q}^*$ of $Q$: We assume a correct and complete algorithm $\mathcal{A}(Q, T_T)$ that finds in an arbitrary typed tree $T_T$ the images of all embeddings of query trees in $\mathbb{Q}^*$. Without modifications, we use this algorithm to find all tree classes belonging to images of data-tree embeddings.

**Lemma 6.3 (Tree-class completeness)** *Algorithm $\mathcal{A}(Q, T_S)$ finds the tree classes belonging to the images of all data-tree embeddings of query trees in the closure of $Q$.*

**Proof:** Let $\mathbb{Q}^*$ be the closure of $Q$, $T_Q \in \mathbb{Q}^*$ be a query tree and $T'_D$ be the image of an embedding $f$ of $T_Q$ into $T_D$. Then $T'_D$ has an embedding $f'$ into $T_S$ that selects the tree class $T'_S$ of $T'_D$ (Lemma 6.1). Since there are embeddings $f$ and $f'$ there is also an embedding $f''$ from $T_Q$ into $T_S$ selecting $T'_S$ (Lemma 6.2). And since $f''$ exists, the image $T'_S$ of $f''$ is selected by algorithm $\mathcal{A}(Q, T_S)$. The same argumentation holds for all query trees in $\mathbb{Q}^*$ and therefore, the lemma holds. □

We will provide an algorithm $\mathcal{A}$ as an adapted version of algorithm `primary` in Section 6.3. However, the lemma together with an algorithm are only the first steps towards a schema-driven query evaluation: If we use a query tree and a schema as input for an algorithm then we will get all tree classes of embedding images – but we are interested in the images itself because we want to find the roots of all results matching the query. We therefore need an "second-level" algorithm $\mathcal{B}$ that finds all embedding images in the data tree given the tree class of those images. Moreover, a query-evaluation algorithm executed against a schema will find all tree classes of embedding images – but will also select included trees of the schema that are not tree classes of embedding images. Thus, algorithm $\mathcal{B}$ must find all embeddings of a particular tree class (completeness), but only embeddings that belong to tree classes (correctness).

**Lemma 6.4 (Reverse embeddings)** *Let $T'_D$ be an included tree of the data tree and $T'_S$ be its tree class selected by embedding $f$. Then $T'_S$ has a root-preserving embedding $f'$ into $T'_D$.*

**Proof:** We prove the lemma by constructing an embedding $f'$. Every node $v_S$ of $T'_S$ is the image of a node $v_D$ of $T'_D$. If $v_S = f(v_S)$ is the image of only one node $v_S$ then we define $f'(v_S) = v_D$. Otherwise, if $v_S$ is the image of several sibling nodes of $T'_D$ then we choose an arbitrary one (say $w_D$) and define $f'(v_S) = w_D$. Every node of $T'_S$ is mapped to a single node in $T'_D$. Therefore, $f'$ is a function. The properties 2, 3, and 4 of Definition 4.1 are equivalences. Therefore, for every pair $u_S, v_S$ holds $label(u_S) = label(f'(u_S))$, $type(u_S) = type(f'(u_S))$, and $(u_S, v_S) \in E'_S \Leftrightarrow (f'(u_S), f'(v_S)) \in E'_D$. It follows that $f'$ is an embedding from $T'_S$ in $T'_D$. □

For the following lemmata and theorems we assume an algorithm $\mathcal{B}$ that takes a set $S_T$ of trees and embeds every tree into a target tree. The algorithm returns the roots of all subtrees of the target tree that contain directly images of exact embeddings of the trees in $S_T$. We say that a subtree contains an embedding image directly if the subtree and the image have the same root. To find all approximate results of $Q$, we have to find all tree classes of the embedding images using algorithm $\mathcal{A}$ and to use the tree classes as input for Algorithm $\mathcal{B}$, which selects the results. The following theorem shows that $\mathcal{B}(\mathcal{A}(Q, T_S), T_D)$ in fact finds the roots of all appropriate results:

**Theorem 6.1 (Completeness)** *$\mathcal{B}(\mathcal{A}(Q, T_S), T_D)$ finds all approximate results of $Q$ in $T_D$.*

**Proof:** Lemma 6.3 states that algorithm $\mathcal{A}(Q, T_S)$ finds all tree classes belonging to images of data-tree embeddings of query trees in the closure of $Q$. Let $T'_S$ a tree class selected by this algorithm. According to Lemma 6.4 there are root-preserving reverse embeddings of $T'_S$ into each of its instances. By assumption $\mathcal{B}(\{T'_S\}, T_D)$ finds the roots of all data subtrees that contain directly images of embeddings of $T'_S$ and thus, among the subtrees selected by $\mathcal{B}(\{T'_S\}, T_D)$ are all subtrees of $T_D$ that directly contain instances of $T'_S$. The same argumentation holds for all tree classes selected by $\mathcal{A}(Q, T_S)$ and therefore, the theorem holds. □

The successive application of the algorithms $\mathcal{A}$ and $\mathcal{B}$ guarantees that the roots of all data subtrees whose distance to $Q$ is less than infinite are found. However, since $\mathcal{A}(Q, T_S)$ finds images of schema embeddings that are not tree classes, we must ensure that algorithm $\mathcal{B}$ finds *only* approximate results of $Q$.

**Theorem 6.2 (Correctness)** *$\mathcal{B}(\mathcal{A}(Q, T_S), T_D)$ finds only approximate results of $Q$ in $T_D$.*

**Proof:** By assumption, algorithm $\mathcal{A}(Q, T_S)$ finds only images of schema embeddings of query trees in the closure of $Q$. Let $T'_S$ be an embedding image selected by $\mathcal{A}(Q, T_S)$. Then there must be a query tree $T_Q$ and an embedding $f$ whose image is $T'_S$. If $T'_S$ is a tree class then it has at least one instance $T'_D$ and thus, there is a root-preserving reverse embedding $f'$ from $T'_S$ into $T'_D$ (Lemma 6.4). The root of the data subtree that directly contains the image of the reverse embedding (and $T'_D$) will be found by algorithm $\mathcal{B}$, which is assumed to be correct. Since $f$ and $f'$ exist, there exists also an embedding $f''$ from $T_Q$ into $T_D$ such that $f'$ and $f''$ have the same image (Lemma 6.2). Because $f''$ exists the root of the data subtree with the same root as $T'_D$ is a result for $Q$. The same argumentation holds for all tree classes selected by $\mathcal{A}(Q, T_S)$, and therefore, the theorem holds. □

Starting in Section 6.3, we present an algorithm $\mathcal{A}$, which finds all second-level queries for a given query and a given schema, and an algorithm $\mathcal{B}$, which finds the roots of data subtrees that contain directly results of second-level queries. However, the number of second-level queries may be exponential with respect to the number of sibling selectors and the number of hierarchy levels of the original query. Our realization of algorithm $\mathcal{A}$ is therefore able to select only the *best* $k$ second-level queries. Furthermore, it computes the embedding cost for each generated second-level query. The following subsection introduces the compaced variant of a schema.

## 6.2 Compacted Schemata

A compacted schema is space efficient representation of a schema. It has the same properties as a schema with respect to the approximate query matching problem but does not store the labels assigned to the nodes. All labels are maintained *only* in type-dependent index structures as described in Section 5.2 on page 8. Furthermore, all leaves that are children of the same node are merged into a single node.

Why is this merging admissible in view of the two-level querying approach described in the previous section? First, all merged leaves are cost-equivalent, which means that they have the same distance to each of their ancestors. Second, we use a slightly modified embedding function that maps a query selector to a schema leaf if the label of the selector is an element of the set of labels assigned to the leaf. Third, an embedding is a function that is not injective. Therefore, each query tree that can be embedded in a schema can also be embedded into its compacted version. Fourth, we "demerge" the merged leaves of second-level queries if they are matched by distinct leaves of a query tree (see Section 6.3.1). It follows that a schema and its compacted version have the same set of embeddable query trees and that the second-level queries derived from both tree variants are identical.

Compacted schemata are encoded using the same numbering scheme as applied to data trees: For each node $v_S$ the value $pre(v_S)$ represents the preorder number of $v_S$ assigned during a depth-first traversal, $bound(v_S)$ is the number of the rightmost leaf of the subtree rooted at $v_S$, $pathcost(v_S)$ is the sum of the insert costs of all nodes along the path from the root to $v_S$, and $inscost(v_S)$ is the insert cost of $v_S$. Note that the node identification by its preorder number is not sufficient in practise. If new documents are added to the database, new nodes may be inserted in the schema, which invalidates the preorder numbering. In our implementation we additionally assign a unique node identifier to each node. The preorder numbers are reassigned after each loading sequence. Figure 7 shows the compacted version of the schema depicted in Figure 5. All labels are displayed in grey color to indicate that they are not stored in the tree. For simplicity, we omit node types in the figure.

How are the path costs of a data tree and its schema related? Recall that a schema preserves all parent-child relationships of its data tree. If $u_D, v_D$ are data nodes and $[u_D], [v_D]$ are their node classes then the path between $u_D$ and $v_D$ (if any) consists of nodes with the same types and labels as the nodes along path between $[u_D]$ and $[v_D]$. It follows that

$$distance(u_D, v_D) = distance([u_D], [v_D])$$

holds for each pair $u_D, v_D$ of data nodes that stand in an ancestor-descendant relationship.

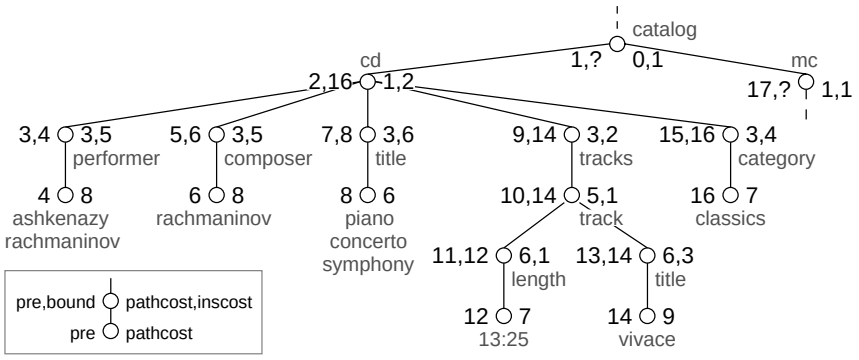



Figure 7: The compacted and encoded schema of the data tree depicted in Figure 3.

| | |
|---|---|
| 13:25 : 12 | catalog : 1 |
| ashkenazy : 4 | category : 15 |
| classics : 16 | cd : 2 |
| concerto : 8 | ... : ... |
| piano : 8 | performer : 3 |
| rachmaninov : 4,6 | title : 7,13 |
| symphony : 8 | track : 10 |
| vivace : 14 | tracks : 9 |
| (a) Index $I_{text}$ | (b) Index $I_{struct}$ |

Figure 8: Indexes of the schema depicted in Figure 7.

To access all schema nodes of a given type and label we use the indexing model developed for data trees (see Section 5.2). Figure 8 shows the indexes of our example schema. The number of occurrences of a given label is typically much smaller in the schema than in the data tree and thus, the postings in schema indexes are short. This allows us to hold some or all indexes in main memory.

## 6.3 Finding the Best $k$ Second-Level Queries

Our objective is to find the images of schema embeddings using algorithm `primary` presented in Section 5. We do not modify the algorithm itself but adapt the definition of lists and change the functions that work on lists: We extend list entries so that they can track not only an embedding cost but also the image of the embedding for which the cost has been calculated. Then we introduce list segments, which are intervals of list entries. Each entry of a segment represents a different embedding image with respect to the same subtree of the schema. Finally, we show how four functions must be adapted to find the best $k$ –or even all– second-level queries.

### 6.3.1 Representing the Images of Schema Embeddings

Recall from Section 5.3 on page 10 that a list is a sequence of entries, where each entry stores information about a data node and an embedding cost. We now extend entries in a way that they represent not only the cost of an embedding of a query subtree but also the the image of that embedding. We define *extended entries* to be a seven-tuple

$$(pre, bound, pathcost, inscost, embcost, label, pointers),$$

where the first five components form a (normal) entry, the component label is the label of the query node that has selected the entry, and the component *pointers* is a set of references to extended entries. Assume that an entry $e$ refers to a schema node $v_S$ that is the match of a query node $v_Q$. If the subtree rooted at $v_S$ has been evaluated then the pointer set of $e$ contains $k$ references to extended entries, where $k$ is the number of children of $v_S$. Each of these $k$ entries represents a schema node that is a match of a child of $v_Q$. Thus, the extended entry $e$ represents the image of the embedding whose cost is $embcost(e)$. Why is the component *label* necessary? A compacted schema does not store the node labels, and it merges all leaves that have the same parent. An extended list entry maintains this missing information. It stores information about the embedding image *and* about the matching query subtree. This information is sufficient to use the embedding image as second-level query as described in Section 6.1. The type information is not stored because only the preorder number and the label of a node are needed to locate all the node instances (see Section 6.4.1).

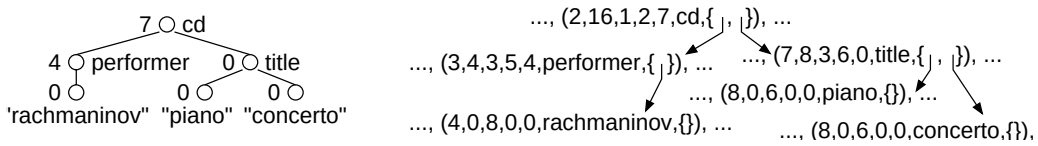



Figure 9: A semi-transformed query tree and the image of its schema embedding.

Figure 9 shows a semi-transformed query included in the expanded query representation depicted in Figure 2(a) on page 8 and the image of its embedding into the schema depicted in Figure 7. The entry component *embcost* in the list belonging to performer has the value 4 because the label is a renaming from composer. The top-level entry has the embedding cost 7, which represents the renaming of composer and the deletion of track. In this example, the embedding costs of the semi-transformed query and its embedding image coincide because no nodes have been inserted. The query leaves "piano" and "concerto" map to the same schema node but can nevertheless by distinguished because the labels are stored as well.

### 6.3.2 List Segments

In Section 5.3 we have defined a list as a sequence of entries, where each entry referred to a different data node. This was reasonable since we were interested in the best embedding per matching data subtree and thus, we have kept only the smallest embedding cost per subtree. Now we want to find the images of the best $k$ approximate embeddings of a query in a schema in order to use them as second-level queries. Two distinct schema embeddings may have have distinct embedding roots, for instance the cd node and the mc node. However, often two distinct embeddings have the same root but map to different included subtrees of the schema. For example, two embedding images may have the common root cd but one of them may contain the composer path whereas the other may contain the performer path. To track the images of the best $k$ embeddings (and their costs) per query subtree and per subtree of the schema, we use *list segments*. A list 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* subtree of the schema. Recall that lists are sorted by the preorder numbers of the list entries. We now additionally require that any list segment is sorted by embedding cost ascendingly.

### 6.3.3 Functions for Extended Lists

In this section, we show that the query-evaluation approach proposed in Section 5 can be extended in a straightforward way to find the best $k$ second-level queries in a schema. We simply have to adapt four of the functions so that they can handle extended lists. The algorithm `primary` itself does not need to be modified. The basic idea is to track not only the *best* embedding but the *best* $k$ embeddings per query subtree and per subtree of the schema. Any list segment whose length is

at most $k$ represents such a sequence of embeddings. All function select the best $k$ combinations of entries from the corresponding segments in the operand lists. To this end, the functions get an additional parameter $k$, which determines the maximal number of embeddings per subtree of the schema. We now sketch the implementation of the four adapted functions:

**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 $\mathit{distance}(e_A, e_D) + \mathit{embcost}(e_D) + c_{edge}$ and its pointer set is initialized with a reference to $e_D$.

**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 $\mathit{embcost}(e_L) + \mathit{embcost}(e_R) + c_{edge}$ and its pointer set is initialized with $\mathit{pointers}(e_L) \cup \mathit{pointers}(e_R)$.

**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 $c_{edge}$.

All other operators (selection, merging, sorting) does not need to be modified: The selection operator creates lists in which each segment has length one; the merging operator accepts only lists with segment length one and merges them to a new list with segment length one; and the sorting operator sorts by embedding cost ignoring segments.

### 6.3.4 Generation of Second-Level Queries

The extended operators can be used by algorithm `primary` like their counterparts for simple lists (see Section 5.4 on page 10). Algorithm `primary` takes the additional parameter $k$ and passes it to the extended operators. If $k$ is infinite then *all* second-level queries are generated. Let $u_Q$ be the root of the expanded representation of a query $Q$. If algorithm `primary` uses the functions for extended lists accesses the indexes of the schema $T_S$ then we have a realization of algorithm $\mathcal{A}$ presumed in Section 6.1:

$$\mathcal{A}(Q, T_S) \stackrel{\text{def}}{=} \texttt{primary}(u_Q, 0, [\,], \infty).^3$$

Typically we are not interested in all second-level queries but in the best $k$ ones in cost-sorted order. The parameter $k$ is passed to every operator to restrict the segment length, which determines the number of embedding images to track per subquery and per subtree of the schema. Segments are

[3] The definition ignores some technical differences between the algorithm $\mathcal{A}$ and `primary`: The former returns a set whereas the latter returns a list. Furthermore, the second-level queries returned by $\mathcal{A}$ are included trees of the schema whereas `primary` returns extended entries. An extended entry represents only the cost-annotated "skeleton" of a second-level query; the inserted nodes are not included. We show in Section 6.4 that a second-level query can be answered using its skeleton only.

sorted by embedding cost and thus, the embedding images belonging to a subtree of the schema are already sorted. However, often several subtrees of the schema contain second-level queries. For example, we may find $k$ second-level queries in the cd subtree and further $k$ second-level queries in the mc subtree of the schema depicted in Figure 5 on page 14. We therefore must apply an outer `sort` operator, which selects the best $k$ second-level queries from all matching subtrees. The expression

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

returns the best $k$ second-level queries for $Q$. All optimizations proposed in Section 5 can be used. In particular, we can apply dynamic pruning based on a given threshold.

## 6.4 Finding the Results of a Second-Level Query

The query-evaluation algorithm, which now uses operators for extended lists, returns a list of second-level queries sorted by embedding cost. A second-level query can be evaluated by a simple algorithm that traverses the data tree and finds exact matches of the query. There is, however, a slight difference to the theoretical setting in Section 6.1: The algorithm `evaluate` does not return "full" embedding images but only "skeletons" of them. More precisely, it returns images that do not represent the inserted nodes, because the costs of the nodes to insert have been derived from the encoding of the schema. Fortunately, it is not necessary to know the nodes implicitly inserted between two skeleton nodes $u_S$ and $v_S$ because all pairs of instances of these nodes have the same distance as $u_S$ and $v_S$ (see Section 6.2). To find the paths of the data-tree that are instances of the path between $u_S$ and $v_S$ we have the know all instances of both nodes and to test for each pair of instances whether they stand in an ancestor-descendant relationship. Again we use the preorder-bound encoding of the data tree to verify those relationships. We first present a secondary index that helps to access all instances of a node class, and present then an algorithm that evaluates the skeleton of a second-level query based on the secondary index.

### 6.4.1 The Secondary Index

To find all instances of a schema node (i.e., a node class), 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. Every structural schema node is uniquely identified by its preorder number. Text nodes in the schema collect all terms at the end of corresponding label-type paths in the data tree. A term is uniquely identified by the preorder number of the text node together with the label of the term. Recall from Section 6.3 that both the preorder number and the label are components of extended list entries. Given a list entry $e$, we construct the key for $I_{sec}$ by concatenating the preorder number and the label: $pre(e)\#label(e)$. Figure 10 shows a part of the secondary index for the data tree depicted in Figure 3 on page 9. For example, the key 4#rachmaninov represents the label rachmaninov of the schema node 4 in Figure 7 and is a component of the posting belonging to label rachmaninov in the text index shown in Figure 8. The posting belonging to key 4#rachmaninov represents all text nodes in the data tree that have the label rachmaninov and are at the end of a label-type path

$$(\text{catalog},\textit{struct}).(\text{cd},\textit{struct}).(\text{performer},\textit{struct}).$$

### 6.4.2 Evaluating a Second-Level Query

Algorithm 3 finds the roots of all exact embeddings of a second-level query (represented by entry $e_1$) in a data tree. The algorithm starts at the query root ($e_1$), traverses the tree in top-down fashion, and evaluates the results bottom up. During the descent it fetches the occurrences of the current query node (Line 1). Then, it evaluates each child of the node separately (Line 2): First, the subtree rooted at the child is evaluated and the embedding roots are stored in list $L_{desc}$ (Line 3). Second, all entries of $L_{anc}$ that have descendants in $L_{desc}$ are selected and appended to a

| | |
|---|---|
| 1#catalog | : (1,?) |
| 2#cd | : (1,15),(16,22) |
| 3#performer | : (3,4),(19,20) |
| 4#ashkenazy | : (4,4) |
| 4#rachmaninov | : (20,20) |
| 6#rachmaninov | : (6,6) |
| ... | : ... |
| 16#classics | : (18,18) |
| 17#mc | : (23,?) |

Figure 10: Index $I_{sec}$ of the data tree shown in Figure 3.

temporary list $L_{temp}$ (Lines 5-7). At the end of the inner loop, $L_{temp}$ is swapped to $L_{anc}$ so that it contains now only entries that have descendants in $L_{desc}$. The same procedure is applied to each child of the current query node so that the returned list $L_{anc}$ consists of all data nodes that are the roots of subtrees that include matches of the whole query subtree. All lists used by the algorithm consist of pre-bound pairs as returned by the secondary index. The topological relationships between the nodes referred to by those pairs can be used to establish the ancestor-descendant relationship in linear instead of quadratic time (see Section 5.4 on page 10).

**Algorithm 3** finds all exact results for a second-level query.

**function** `secondary`($e_1$)

**input**: $e_1$ – a second-level query represented by entry $e_1$,
**output**: $L_A$ – the list of results for $e_1$.

1: $L_A \leftarrow I_{sec}(\, pre(e_1)\#value(e_1)\,)$
2: **foreach** entry $e_2$ referred to by $pointers(e_1)$ **do**
3: $L_D \leftarrow$ `secondary`($e_2$)
4: $L_T \leftarrow [\,]$
5: **foreach** entry $e_3$ of $L_A$ **do**
6: **if** $e_3$ has a descendant in $L_D$ **then**
7: append $e_3$ to $L_T$
8: $L_A \leftarrow L_T$
9: **return** $L_A$

## 6.5 An Incremental Algorithm for the Best-$n$-Pairs Problem

So far, we have seen how to construct second-level queries and how to find the roots of all results for each second-level query. Algorithm 4 integrates both parts. At Line 1 the best $k$ second-level queries are created and sorted by embedding cost. Then, the queries are evaluated by function `secondary`. Each preorder-bound pair retrieved by this function is the root of a result; the embedding cost of $Q$ for this document is determined by the entry $e_1$ taken from $L_{prim}$. Because any result may be matched by several second-level queries (with different embedding costs) the algorithm must keep track of documents already retrieved. Line 6 sketches the test against the history of the preorder numbers of the roots of retrieved document. In practice, we use a hash function to implement this test. Algorithm 4 fulfills the requirements of a correct and complete retrieval algorithm requested in Section 6.1:

$$\mathcal{A}(\mathcal{B}(Q, T_S), T_D) \stackrel{\text{def}}{=} \texttt{retrieve}(Q, \infty).^4$$

[4] Again there are some technical differences: $\mathcal{A}(\mathcal{B}(Q, T_S), T_D)$ retrieves a set of nodes whereas Algorithm 4 retrieves a cost-sorted list of pairs $(pre, embcost)$, where $pre$ is the number of the preorder number of a data node and $embcost$ is the distance between $Q$ and the result rooted at this node.

**Algorithm 4** retrieves the root-cost pairs for the best $k$ second-level queries.

**algorithm** `retrieve`$(Q, k)$

**input**: $Q$ – a query,
$k$ – the number of second-level queries to generate for $n$ results,
**output**: the root-cost pairs of the results for the best $k$ second-level queries.

```
1: let u_Q be the root of the expanded representation of Q
2: L_P ← sort(k, primary(u_Q, 0, [ ], k))
3: foreach entry e_1 of L_P do
4:    L_S ← secondary(e_1)
5:    foreach e_2 of L_S do
6:       if pre(e_2) has not yet output then
7:          output (pre(e_2), embcost(e_1))
```

Algorithm 4 provides a realization of the two-level retrieval approach discussed in Section 6.1. However, our objective is to retrieve the *best* $n$ results for a query. If we knew beforehand how many second-level queries were necessary to find $n$ results we could determine $k$ from $n$ and pass it to Algorithm 4. Unfortunately, there no strong correlation between $k$ and $n$; some second-level queries may retrieve many results, some may not find any result at all. Therefore, we must *guess* the initial value of $k$ and increment it by a delta $\delta$ if the first $k$ second-level queries have not retrieved enough results. Fortunately, the increase of $k$ does not invalidate the previous results: The list $L_{prim}$ returned by algorithm `evaluated` for a certain $k$ is a prefix of the list $L'_{prim}$ returned for a $k' \geq k$.

**Algorithm 5** retrieves incrementally the best $n$ root-cost pairs for a query.

**algorithm** `retrieve_incremental`$(Q, k, n, \delta)$

**input**: $Q$ – a query,
$k$ – initial guess for the number of second-level queries to generate,
$n$ – the number of requested results,
$\delta$ – increment for $k$ if the number of queries is not sufficient,
**output**: the root-cost pairs of the best $n$ results for $Q$ in sorted order.

```
1: let u_Q be the root of the expanded representation of Q
2: k_prev ← 0
3: while the number of output documents is less than n do
4:    L_P ← sort(k, primary(u_Q, 0, [ ], k))
5:    erase the first k_prev entries from L_P
6:    execute Lines 3-7 of Algorithm 4
7:    k_prev ← k; k ← k + δ
```

Algorithm 5 shows an incremental version of Algorithm 4. After generating the execution plan for $Q$ (Line 1) the algorithm repeats the creation and evaluation of second-level queries until $n$ results have been found. Note that the algorithm terminates even if $n$ is infinite because no second-level query is created twice and because the number of included trees in the schema is finite. The algorithm erases at any step the prefix of all second-level queries that have already been evaluated (Line 5) and adds $\delta$ to the number $k$ (Line 7). All second-level queries remaining in $L_{prim}$ are evaluated as described for Algorithm 4.

What are appropriate values for $k$ and $\delta$? In our experiments we could not find an initial value for $k$ that was satisfying for all queries and data trees. This is not surprising because all horizontal dependences of the data tree are lost in the schema, and thus, we cannot say beforehand whether two query siblings have common matches in a data subtree. In the most cases, we got the fastest answers if we set $k = n$. Also, it proved to be successful if we incremented $k$ non-linearly. We set

$\delta = k$ and doubled the value of $\delta$ at each iteration step.

## 6.6 Complexity Analysis

Recall from Section 5 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 the maximal number of repetitions of a label along a path. In the following, we use the letters $s_s$ to denote the selectivity in the schema and $s_d$ to denote the selectivity in the data tree, i.e., the maximal number of instances of a node class. The time complexity of the functions modified in Section 6.3 rises by the factor $k^2 \cdot \log k$, which is the time needed to compute segments of size $k$ from the operand lists. Therefore, the maximal 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 a second-level query is bound by $O(s_d \cdot m)$, where $m$ is the number of nodes in a second-level query. The maximal time needed for a single iteration of the algorithm shown in in Figure 5 is

$$O(k \cdot (n^2 \cdot r \cdot s_s \cdot l \cdot k \cdot \log k + s_d \cdot m)).$$

Note that all parameters of the formula are typically small numbers. This is true even for $s_s$ and $s_d$ because $s_s$ cannot exceed the number of schema nodes and $s_d$ is bound by the maximal number of equal paths in the data tree.

## 6.7 Optimizations

The schema-driven incremental query evaluation proposed in this section guarantees that we get the best $n$ answers without computing the whole set of approximate results. However, our approach as it is summarized in Algorithm 5 has still some deficiencies:

**Query similarity.** Each second-level query represents an included tree of the schema. Many queries differ only in one path or even one node. Because the selectivity of a second-level query is typically high, we must often choose a large $k$ to find $n$ results.

**Redundant computing.** Algorithm 5 evaluates each second-level query independently. Since many queries have common substructures, the algorithm performs the same computation steps several times.

In this section, we propose a technique that effectively reduces the number of second-level queries but increases the evaluation time of a single query. Then, we show how caching can be used to avoid the redundant evaluation of common query substructures.

### 6.7.1 Merging of Second-Level Queries

A second-level query is the image of an approximate embedding of an approXQL query in a schema. The operators for extended lists track the images of the best $k$ embeddings per subquery and per subtree of the schema regardless whether the have common substructures. Consider the query

```
cd[composer/"rachmaninov" and title/"concerto"]
```

and assume that we are allowed to change `composer` into `performer`, `title` into `category`, and the word `concerto` into `sonata` or `symphony`. Assume further that all possible parent-child combinations existed once in a compacted schema. Figure 11 shows a detail of a schema for which our assumption holds. If no deletions were permitted then Algorithm 5 would generate 12 distinct second-level queries (supposed that $k \geq 12$). The selectivity of a second-level query is typically high and many of them may not find a result in the data tree at all. Thus, we must either use a large initial value for $k$ or increment it several times in order to find $n$ results.

Our objective is to reduce the number of created second-level queries and to increase the number of results per query. During the construction of the second-level queries we merge subqueries that have different structure but the same embedding cost. This happens especially ($i$) if the renaming

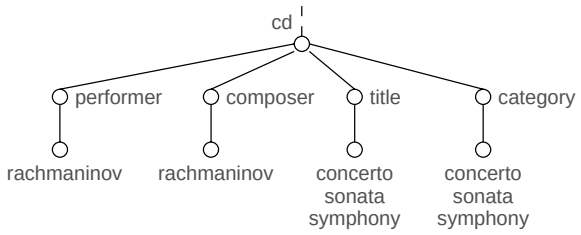



Figure 11: A detail of a compacted schema.

costs are equal for several alternative labels of a selector or (*ii*) if a leaf selector maps to several leaves of the schema that all have the same distance to a matched ancestor node. We connect such cost-equivalent subtrees of second-level queries by "`or`" operators. A second-level query that consists of both conjunctive and disjunctive parts is called *hybrid second-level query*. Hybrid second-level queries are evaluated by an extended version of Algorithm 3. This extended algorithm processes a disjunction of query subtrees by merging the data-tree matches of each group member into a single list. This increases the execution time of a single second-level query but effectively reduces the number of queries. If we assume that each renaming of `concerto` has the same cost then we can create the disjunction of the matches of `sonata` and `symphony`. This reduces the number of second-level queries from 12 to 6. The more cost-equivalent renamings exist and the more cost-equivalent matches in the schema exist the more effective is this query merging.

### 6.7.2 Caching of Intermediate Results

Algorithm 5 creates a list of $k$ second-level queries and evaluates them successively using the function `secondary`. Many second-level queries have common subtrees. For example, among the $k$ queries there may be several queries that differ in one label, whereas the remaining structures are identical. The repeated evaluation of identical subtrees is redundant and should be avoided. We use a straightforward solution for this problem: We cache intermediate results computed for query subtrees and use the cached results whenever we detect a subtree that has already been evaluated. There are, however, two subtle difficulties: First, if we have no knowledge about the subsequent queries then we must cache the results computed for every subtree of every query. Second, to determine whether a subtree has already been evaluated we must compare the subtrees. We avoid both the "blind" caching and the expensive tree comparison by tracking common subtrees during the construction of the second-level queries. Since we use references to entries instead of copying them (see Section 6.3.3) we can easily identify common subtrees. Each node of the directed acyclic graph of second-level queries that has two or more parents is a candidate for caching. We simply annotate those nodes with unique numbers in order to use them as keys of the cache.

# 7 Experiments

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

## 7.1 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 [ANZ01]. 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 three document collections that each consist 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 so that the most frequent word occurs twice as many as the second-most word, and so on. We call the number of elements in a collection the *collection size*. The size of a collection is equal to the number of structural nodes in the data tree constructed from the collection (see Section 3). Similarly, the *schema size* is the number of structural nodes in the schema of a collection (see Section 6). The three collections differ only in the schema sizes as the table shows:

| | collection 1 | collection 2 | collection 3 |
|---|---|---|---|
| schema size | 100 | 1,000 | 10,000 |
| ratio of schema size to collection size | 1 : 10,000 | 1 : 1,000 | 1 : 100 |

Despite the different schema sizes, the average selectivity per element name is 1/10,000 in all three collections. We ensure these equal average selectivities by using the same 100 element names several times in a schema. The repeated names are placed randomly; they may appear along the same path or in different subtrees of the schema.

All queries used in our experiments are produced by a simple generator for approXQL queries. The generator expects a query pattern, which determines the structure of the query. A query pattern consists of name templates, term 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 terms and names in the query. All costs are chosen randomly from the interval [1..10]. 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.

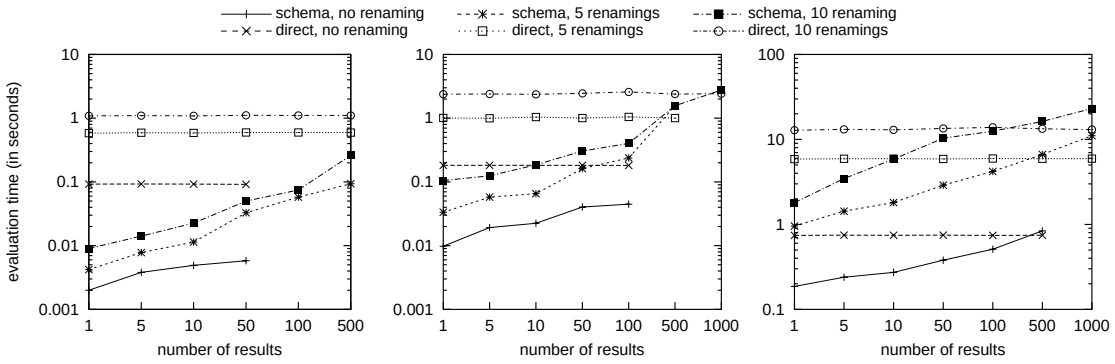
All tests have been carried out on a Pentium III with 450 MHz and 256 MB memory running Linux as operating system. Our system is implemented in C++ on top of the Berkeley database [BER00].

## 7.2 Test Results

Our tests yielded two main results: First, the schema-driven query evaluation is much faster then the direct evaluation if $n$, the number of results, is small. For some queries the schema-based algorithm is faster even if $n$ is infinite (that is, if all results are requested). Second, the response time the schema-based algorithm independent of the schema size.

The three diagrams depicted in Figure 12 illustrate the first result. The diagrams show the evaluation times of the three query patterns in the data tree that represents collection 1. The x-axis of each diagram denotes $n$, the number of desired results for the queries; the y-axis shows the evaluation time. Note, that the y-axis has a logarithmic scale. Each diagram consists of six curves that describe the evaluation time of the same query pattern with respect to different numbers of renamings per node. To each number of renamings $(0, 5, 10)$ belong two curves: The first one, labeled "schema", shows the evaluation time of the schema-based algorithm. The second one, labeled "direct", displays the time needed for the direct query evaluation. Any point in the diagrams is the mean of the evaluation time of 10 queries randomly generated for the same pattern.

Figure 12(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. In fact,



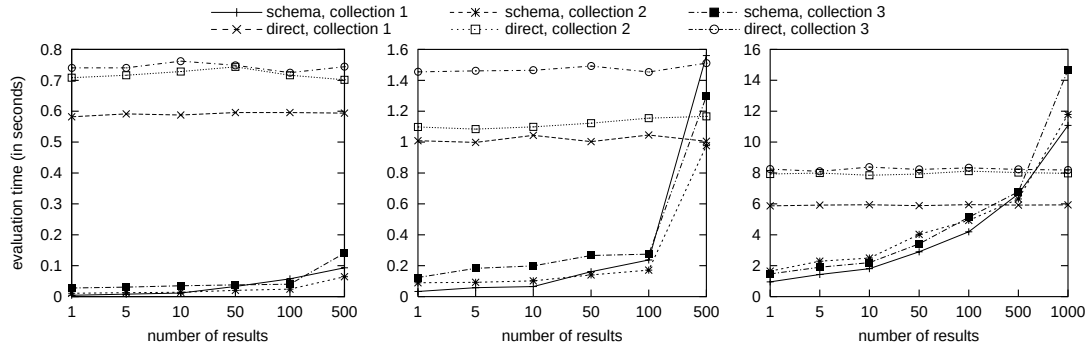

(a) Query pattern 1 (b) Query pattern 2 (c) Query pattern 3

Figure 12: Evaluation time of the three query patterns with respect to collection 1.

the schema-based evaluation is still faster than the direct variant if we do not allow the deletion of nodes. There are two reasons for this behavior: First, all second-level queries generated by the algorithm `primary` have at least one embedding in the data tree (each second-level query corresponds to a label-type path). Second, the algorithm `secondary` uses path-dependent postings. A path-dependent posting stores the matches of a label with respect to its position in the label path. In contrast, the algorithm for the direct evaluation uses postings that contain all matches of the label. The shorter postings of the schema-based approach result in much faster operations on lists.

Figure 12(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 12(c)). For small values of $n$ and few renamings, however, the schema-driven algorithm is still fast and outperforms the direct evaluation.




(a) Query pattern 1 (b) Query pattern 2 (c) Query pattern 3

Figure 13: Evaluation time of the query patterns with respect to different schema sizes.

Figure 13 presents the second main result. The three diagrams show the evaluation times of the three query patterns with respect to three different schema sizes. (We only show the evaluation times of the queries that have five renamings per label. However, all other queries behave similarly.) Consider, for example, the diagram depicted in Figure 13(b). The three curves representing the schema-based algorithm have almost the same gradient – although the schema sizes of the three collections differ largely. The time for the direct evaluation increases slightly for the collections 2 and 3, because they have repeated labels along the paths. In fact, we could conclude

from additional experiments that the *only* data parameters that affect the performance of the algorithms are the selectivity per label and the number of repeated labels along a path.

# 8 Related Work

Our work has three related areas: distance metrics for trees, flexible queries for semistructured data, and information-retrieval extensions for XML query languages.

Several measures for the similarity of trees have been developed [Tai79, JWZ94]. 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 takes into account the different semantics of roots, leaves, and inner nodes with respect to XML data.

The problem of finding the minimal edit or alignment distance between unordered trees is MAX SNP-hard [AG97]. Even the problem of including a query tree into a data tree is NP-complete [Kil92]. The complexity results suggest that these unordered tree-similarity measures cannot be used for querying trees. The ordered variant of tree edit as well as some restricted forms of edit distances have polynomial time complexity. However, the algorithms solving these problems touch every data node, which leads to inadequate answering times for large databases.

The emerge of semistructured data has triggered considerable research concerning flexible queries as a means to cope with the structural heterogeneity of the data. Almost all query languages for semistructured data and XML support regular path expressions, which allow to specify alternative paths through the data graph and to skip certain subgraphs. Although regular path expressions give some additional flexibility, the also require a considerable knowledge about the data. The user must at least know that some subgraphs must be skipped, that alternative paths exist, and how they look like. Kanza et. al. investigate flexible query matchings that require less knowledge about the data. In [KNS99], they propose a query matching that allows partial matches. In a later work [KS01], they investigate matchings where path-connected query nodes may be permuted. The partial matching is similar to the deletion of nodes in our approach; the permutation of nodes has no correspondence in out model. Neither regular path expressions (with the exception of [FFS01]) nor the models by Kanza et. al. valuate the closeness between the query and its results.

To the best of our knowledge, XXL [TW00], ELIXIR [CK01], and XIRQL [FG01] are the only XML query languages that support result ranking. Both XXL and ELIXIR add a similarity operator to a subset of XML-QL [DFF$^+$98]. In XXL, the similarity operator can be applied to element names as well as to text sequences. The query processor searches for matches similar to the name or text specified and assigns probabilities to the matches. The probabilities are combined to obtain a single score. ELIXIR is comparable with XXL but additionally supports similarity joins. XIRQL incorporates the notion of term weights and vague predicates into XQL. The content and structure of XML documents are mapped to facts and rules of an intensional probabilistic logic [RÖ99]. None of the three languages allows partial structural matches.

# 9 Conclusion and Future Work

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 approach can be easily 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 saw, that the schema-driven query-evaluation outperforms the direct evaluation if $n$ is smaller than the total number of results. However, the main advantage of the schema-based

approach is the incremental retrieval: Once the best $k$ second-level queries have been generated, the queries can be evaluated successively, and the first results found can be sent immediately to the user. If many or all approximate results for a query are requested, the direct evaluation should be preferred.

The response times of both algorithms rise if the query contains many conjunctions and if each query node has many renamings. However, the response times increase for different reasons: The algorithm for direct evaluation becomes slower simply because the number of function calls rises. In contrast, the performance of the schema-based approach decreases because several second-level queries have no results. We plan to investigate schemata that provide more information about horizontal dependencies in the data tree. Using this additional information, the query processor can reject non-matching queries at an early stage.

## References


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

[ANZ01] A. Aboulnaga, J.F. Naughton, and C. Zhang. Generating synthetic complex-structured XML data. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, pages 79–85, Santa Barbara, USA, May 2001.

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

[BER00] The Berkeley Database. Sleepycat Software Inc., Lincoln, MA, 2000. http://www.sleepycat.com/.

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

[CK01] T.T. Chinenyanga and N. Kushmerick. Expressive retrieval from XML documents. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New Orleans, USA, September 2001.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. W3C Note, August 1998. http://www.w3.org/TR/NOTE-xml-ql.

[FFS01] S. Flesca, F. Furfaro, and Greco S. Weighted path queries on web data. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, pages 7–12, Santa Barbara, USA, May 2001.

[FG01] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 172–180, New Orleans, USA, September 2001.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured data. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, pages 436–445, Athens, Greece, August 1997.

[JWZ94] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Combinatorial Pattern Matching*, pages 75–86, June 1994.

[Kil92] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Finland, November 1992.

[KNS99] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS)*, pages 227–236, May - June 1999.

[KS01] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of the 20th Symposium on Principles of Database Systems (PODS)*, Santa Barbara, USA, May 2001.

[R̈99] T. Rölleke. *POOL: Probabilistic Object-Oriented Logical Representation and Retrieval of Complex Objects — A Model for Hypermedia IR.* PhD thesis, University of Dortmund, May 1999.

[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.