FREIE UNIVERSITÄT BERLIN

ApproXQL: Design and Implementation of an Approximate Pattern Matching Language for XML

Torsten Schlieder

B 01-02
May 2001

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B · INFORMATIK

ApproXQL: Design and Implementation of an Approximate Pattern Matching Language for XML

Torsten Schlieder*

Freie Universität Berlin

schlied@inf.fu-berlin.de

Abstract

We introduce the simple query language approXQL, which supports hierarchical, Boolean-connected query patterns. The interpretation of approXQL queries is founded on cost-based query transformations: The total cost of a sequence of transformations measures the similarity between a query and the data and is used to rank the results. We describe in detail the implementation of the approXQL query processor, which uses an expanded query representation and sophisticated indexes to compute all results of a query in polynomial – typically sublinear – time with respect to the database size.

1. Introduction

An XML query engine should retrieve the best results possible. If no exact matching documents are found, results *similar* to the query should be retrieved and *ranked* according to their similarity.

The problem of similarity between keyword queries and text documents has been investigated for years in information retrieval [BR99]. Unfortunately, the models developed in this community cannot be directly applied to XML documents, since they

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

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

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

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

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

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


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

will neither find CDs with a *track* title "Piano concerto" nor CDs of the *category* "Piano concerto" nor concertos *performed* by "Rachmaninov", nor other sound storage media than CDs with the appropriate information. The query will also not find CDs where only one of the specified keywords appears in the title. Of course, the user can pose a query that exactly matches the cases mentioned – but she must know beforehand that such similar results exist and how they are represented. Moreover, since all results of the expanded query are treated equally, the user still cannot express her preferences.

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

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

The paper is organized as follows: In the next section, we introduce the syntax of the approXQL query language. In Section 3, we show how both queries and XML documents can be interpreted as labeled trees. Based on the similarity between a query tree and a data tree, we introduce the semantics of our query language in Section 4. Section 5 presents additional syntactical constructs of the approXQL query language that allow the user to express her preferences. In Section 6, we explain in detail the design of a query-evaluation algorithm for approXQL. In Section 7, we describe the prototypical implementation of our query engine. We review related work in Section 8 and conclude in Section 9.

2. The ApproXQL Query Language

We present a simple query language called approXQL that is inspired by XQL [RLS98]. ApproXQL queries are tree-shaped search patterns with *existence* semantics: A document part matches a query if the names and keywords of the query exist and fulfill the desired relationships. ApproXQL does not have language constructs for transforming documents and for defining results. Like in XQL, a query result is just a document part matched by the search pattern. Informally, approXQL consists of the following syntactical constructs:

- name selectors like "cd", "composer", and "title",
- the containment operator "`[]`",
- the `text()` selector, which matches words and phrases in text sections and attribute values,
- the operator "=", which compares its left-hand operand with a string constant, and
- the logical connectors "`$and$`" and "`$or$`".

In addition, parentheses can be used to specify operator precedences in Boolean expressions. The following query selects CDs appeared 2001 that contain works composed by Rachmaninov:

```
cd[year[text() = "2001"] $and$ composer[text() = "rachmaninov"]]
```

Note, that the user does not need to know how the year 2001 is modeled in the original documents because the text() operator matches both cases. In contrast to the corresponding operator in XQL, the `text()` operator in approXQL always selects *substrings*. If its right-hand operand consists of a single word only (and not a phrase) then the stem of the word is derived.

The "`$or$`" operator can be used to specify alternative alternative paths through the document:

```
cd[year["2001"] $and$ composer["rachmaninov" $or$ "prokofiev"]]
```

The above query shows that the `text()` selector can be omitted. However, this abbreviation is not permitted if another operator than text() is used (see Section 5). We do not require that the inner parts of the query select words or phrases. The following query matches all CDs by Rachmaninov that have a review or a description:

```
cd[composer["rachmaninov"] $and$ (review $or$ description)]
```

The simple core syntax of approXQL will be adapted frequently in the rest of the paper: In the next section, we show how approXQL queries are broken up into sets of conjunctive query trees. Section 4 presents the semantics of approXQL queries. In Section 5 we shall see how the default semantics can be adapted to the user's needs. The complete syntax of approXQL can be found in Appendix A.

3. Tree Interpretation of 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 and their Properties

A *rooted tree* is a structure $\mathcal{T} = (N, E, root(\mathcal{T}))$ that consists of a finite set of nodes N, a finite set of edges E, and a node $root(\mathcal{T}) \in N$ that forms the *root node* of $\mathcal{T}$. With $|N|$ we denote the *number of nodes* in $\mathcal{T}$. The set of edges is a binary relation on N where each pair $(u, v) \in E$ establishes a relationship between two nodes of N. We say that u is the *parent* if v, and v is a *child* of u. E must satisfy the following conditions:

1. The root has no parent.
2. Every node of the tree except the root has exactly one parent.
3. All nodes except the root are reachable from the root.

A *path* in a tree is a sequence of nodes $u_1, u_2, \ldots, u_n$ such that for every pair u_i, u_{i+1} of consecutive nodes there is an edge $(u_i, u_{i+1}) \in E$. The path starts from node u_1, ends at node u_n, and has the *length* n. We define the predicate $path(u, v)$ that returns *true* iff $u \neq v$ and it exists a path that starts at u and ends at v. The *height* of the tree is defined as the length of the longest path starting at the root node.

We refer to the parent of a node v using the notation $parent(v)$. The nodes of a tree that have a common parent u are called *children* of u, denoted by

$$children(u) = \{v \in N \mid (u, v) \in E\}.$$

We apply an arbitrary but unique numbering to the children of n and use the notation $child_i(u)$, $1 \leq i \leq |children(u)|$ to refer to the ith child of u. A node without children is a *leaf node*. Nodes that are not leaf nodes are called *inner nodes*.

A rooted tree $\mathcal{T} = (N, E, root(\mathcal{T}))$ in which for every node $u \in N$ a label is defined, is called *labeled rooted tree*. We use the notation $label(u)$ to refer to the label of u. Different nodes of $\mathcal{T}$ may have the same label. Trees for which the property

$$\exists u, v \in N : path(u, v) \wedge label(u) = label(v)$$

holds are called *recursive trees.*

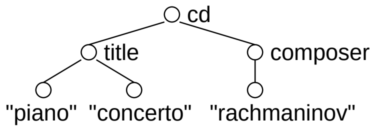
In order to model both text and structural parts of XML documents in the same tree, we define *node types.* Every node of the tree has either type *text* or type *struct.* We use the notation $type(u)$ to refer to the type of u. Note, that only leaf nodes may have type *text.*

3.2. Tree Interpretation of ApproXQL Queries

An approXQL query is a *conjunctive* query if it does not contain `$or$` connectors. A conjunctive query can be interpreted as tree in a simple and intuitive manner: Each `text()` selector is mapped to a leaf nodes of type *text*; each name selector is represented as node of type *struct.* The containment operator is interpreted as parent-child relation in the tree. If the containment expression includes `$and$` operators then the conjunctive normal form of the expression is created. Each operand of the conjunctive normal form is added as child to the containing node. Figure 1(b) shows the tree interpretation of the approXQL query depicted in Figure 1(a).

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

(a) Conjunctive approXQL query



(b) Query tree

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

To represent a conjunctive query tree formally, we use the notation introduced in Section 3.1: A conjunctive query tree is a tuple $\mathcal{Q} = (M, E, root(\mathcal{Q}))$, where M is the set of nodes, E is the set of edges, and $root(\mathcal{Q})$ is the root of the tree.

A query that contains `$or$` operators is broken up into into a set of conjunctive queries, which is called *separated query representation.* The construction of the separated representation of a query is straightforward: Starting at the outermost containment operator, the disjunctive normal form of the Boolean expression enclosed by the containment operator is derived. Each conjunct, together with the toplevel part of the query, forms a new query. The set of all queries that can be created that way is used as input of a new iteration of the normalization algorithm. The procedure ends if no `$or$` nodes remain. If k is the number of `$or$` operators in the original query then the set of conjunctive queries has 2^k elements. As an example, consider the query

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

Its separated representation is the following set of conjunctive queries:

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

We shall see in Section 4 how the set of conjunctive queries derived from a user-provided query is interpreted in order to get a valuation of the query results. In Section 6, we introduce a technique to compute all results of an approXQL query *without* the explicit construction of the separated query representation.

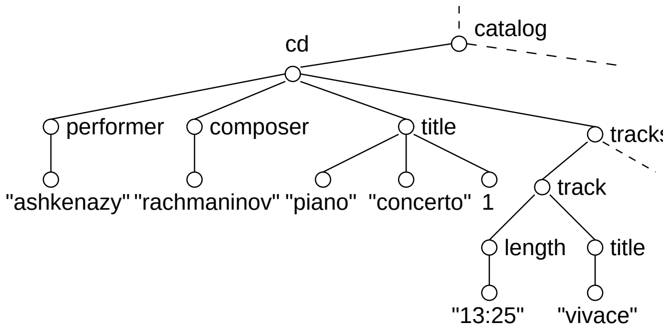
3.3. Modeling and Normalization of XML Documents

We model XML documents as labeled trees, neglecting the semantics of links. Elements are mapped to nodes of type *struct.* The name of the element is used as node label. Text sequences are broken up into words, which are stemmed and converted to lower case. For each word, a leaf node of type *text* is created and labeled with the word. All nodes belonging to a text sequence are added as children to the node that represents the element containing the text.

Attributes are modeled as trees of height two: The attribute name is mapped to a node of type *struct*. This node is added as child to the node that represents the element the attribute belongs to. The attribute value is broken up as described for text sequences, and the words are mapped to *text* nodes. These nodes form the children of the node representing the attribute name.

```
<catalog>
  <cd>
    <performer>Ashkenazy</performer>
    <composer>Rachmaninov</composer>
    <title>Piano concerto 1</title>
    <tracks>
      <track length="13:25">
        <title>Vivace</title>
      </track>
      ...
    </tracks>
  </cd>
  ...
</catalog>
```

(a) XML document



(b) Data tree

Figure 2: Mapping of an XML document to a normalized data tree.

Figure 2 shows the mapping of an example XML document to a normalized tree. We add a new node with a unique label to the collection of document trees and establish an edge between this node and the root of every document tree. The tree representing all document trees is called *data tree*.

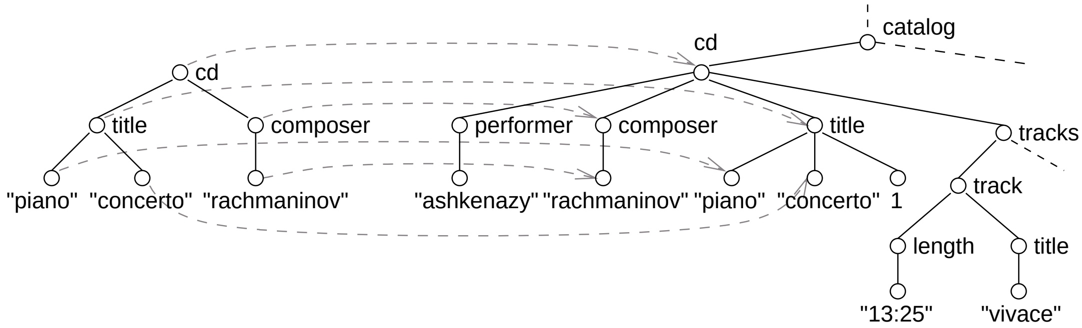
The formal representation of a data tree is a tuple $\mathcal{D} = (N, F, root(\mathcal{D}))$, where N is the set of nodes, F is the set of edges, and $root(\mathcal{D})$ is the root of the data tree.

4. Querying by Approximate Tree Matching

In this section, we introduce the semantics of evaluating an approXQL query. We first define an embedding function that maps a conjunctive query tree to a data tree. The embedding is *exact* in a sense that all labels of the query occur in the result, and that the parent-child relationships of the query are preserved. Starting in Section 4.2, we introduce our approach to find *similar* results to the query. We transform a conjunctive query tree query by inserting, renaming, and deleting nodes. Each basic transformation has a cost; the summary cost of a sequence of basic transformations is assigned to the query results and used to rank them by increasing cost.

4.1. The Tree-Embedding Formalism

We have shown in Section 3 how to interpret both the data and the query as labeled trees with a single node type. With this interpretation, the problem of answering a query can be mapped to the problem of embedding a query tree in a data tree. Our definition of tree matching is inspired by the *unordered path inclusion problem* introduced 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 think that ignoring the order of siblings is favorable or even necessary for querying XML data because the ordering of the elements or keywords may be inconsistent. Even if it were consistent the order might not be known to the user. However, using the language extensions described in Section 5, the user can enforce ordered embeddings.



(a) Query tree (b) Part of an XML tree

Figure 3: Embedding of a query tree in a data tree.

We discard the injectivity property of the path inclusion problem in order to get a function that is efficiently computable[1]:

Definition 1 (Embedding) *Let $\mathcal{Q} = (M, E, root(\mathcal{Q}))$ be a query tree and $\mathcal{D} = (N, F, root(\mathcal{D}))$ be a data tree. A mapping f from the set M of query nodes into the set N of data nodes is called an embedding iff for all $u, v \in M$ the following conditions hold:*

1. $u = v \Rightarrow f(u) = f(v)$ *(f is a function),*
2. $label(u) = label(f(u))$ *(f is label preserving),*
3. $type(u) = type(f(u))$ *(f is type preserving),*
4. $(u, v) \in E \Rightarrow (f(u), f(v)) \in F$ *(f is parent-child preserving).*

We call a data node $u = f(root(\mathcal{Q}))$ that is mapped to the query root an *embedding root.* The data subtree that is anchored at an embedding root is a *result.* Note, that for a fixed query tree and a fixed data tree, several results may exist, and several embeddings may lead to the same result. Figure 3 shows an embedding of a query tree into a data tree.

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.*

Definition 2 (Basic Query Transformation) *Each modification of a conjunctive query tree by inserting a node, deleting a node, or renaming the label of a node is a basic query transformation.*

Each basic transformation has a cost, which is provided by a domain expert and/or the user:

Definition 3 (Basic cost) *A basic cost is a natural number.*

In Section 4.4, we shall discuss several possibilities to assign costs to basic transformations.

In contrast to the tree edit distance [Tai79], we do not allow arbitrary sequences of insert, delete, and rename operations. We restrict the basic transformations in order to generate only queries that have an 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. We shall see in Section 6 that all results can be computed in polynomial time with respect to the number of nodes in the data tree. In the following subsections, we introduce the three types of basic query transformations in more detail.

[1] If we kept the injectivity property of the embedding function then we would run into a "complexity trap": The relaxation of the parent-child-relationship to an ancestor-descendant-relationship, which we will introduce in Section 4.2.1, would lead to the *unordered tree inclusion problem* that is NP-complete [Kil92].

4.2.1. Node Insertion: Moving to a more Specific Context

A node insertion creates a query that expects the matches of a query subtree in a *more specific context*. We only allow the insertion of inner nodes, that is, every edge can be replaced by a node with an incoming and an outgoing edge. It is not allowed to add a new query root, new leaf nodes, or inner nodes with more than one outgoing edge.

Definition 4 (Insertion) *Let $\mathcal{Q} = (M, E, root(\mathcal{Q}))$ be a query tree, $(u, w) \in E$ be an edge, and $v \notin M$ a node. An insertion of v between u and w is a transformation of $\mathcal{Q}$ to $\mathcal{Q}' = (M', E', root(\mathcal{Q}'))$ such that*

$$\begin{aligned} M' &= M \cup \{v\}, \\ E' &= E \setminus \{(u, w)\} \cup \{(u, v)\} \cup \{(v, w)\}. \end{aligned}$$

Cost based node insertions can be considered as a counterpart to the "//" operator of XQL. There are, however, two major differences: First, the user must explicitly use the XQL operator "//" instead of "/" in order to skip an arbitrary number of data nodes. To use this operator, the user must *know* that some nodes must be skipped in order to get the desired results. In our approach, nodes are inserted automatically in order to find an embedding of the query tree. The user does no need to know that some nodes have to be skipped. Second, in our approach the distance between two nodes is measured by the summary cost of the applied insert operations. Using this summary cost we penalize the transition to a more specific context. For instance, the insertion of a node labeled tracks and a node labeled track between the nodes cd and title of the query in Figure 3(a) switches from the more general context *compact disc title* to the more specific context *track title*.

4.2.2. Node Deletion: Moving to a More General Context

The deletion of query nodes is founded on two concepts: vague containment and vague conjunction. Vague containment is based on the observation that the hierarchy of an XML document typically models a containment relationship. The deeper an element resides in the document tree the more specific is the information it describes. For example, the element length describes the length of a track whereas the element track describes the entire track (including its length). Similarly, the element composer describes the composer of the tracks included in the CD whereas the top-level element cd describes the entire CD.

The deletion rule of vague containment allows the deletion of inner query nodes *bottom up*. Assume that a user searches for tracks with the title "concerto". Vague containment allows to remove the node title in order to move to the more general context track in which the keyword "concerto" is searched. Then, the nodes labeled track and tracks (in this order) may be removed from the query. The most general context of this query is an entire CD, in which the word "concerto" may appear as category identifier or title, respectively.

Definition 5 (Deletion of inner nodes) *Let $\mathcal{Q} = (M, E, root(\mathcal{Q}))$ be a query tree, and $v \in M$ be an inner node such that all children of v are leaves. The deletion of node v from M is a transformation of $\mathcal{Q}$ to $\mathcal{Q}' = (M', E', root(\mathcal{Q}'))$ such that*

$$\begin{aligned} M' &= M \setminus \{v\}, \\ E' &= E \setminus \{(u, w) \mid (u, w) \in E \wedge (u = v \vee w = v)\} \cup \{(u, w) \mid (u, v) \in E \wedge (v, w) \in E\}. \end{aligned}$$

Note, that this definition does not allow the deletion of the query root or the deletion of the leaf nodes. The deletion of leaves is the topic of vague conjunction.

Vague conjunction adopts the concept of "coordination level match" [BR99], which is a simple querying model that establishes ranking for queries of and-connected search terms. 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. Let u be an inner query node such that n of its children are leaves of the query tree. Vague conjunction allows the deletion of at most $n - 1$ of those children.

Definition 6 (Deletion of leaf nodes) *Let $\mathcal{Q} = (M, E, root(\mathcal{Q}))$ be a query tree, and $v \in M$ be a leaf node such that the parent of v has at least two children that are leaves (including v). The deletion of a node v from M is a transformation of $\mathcal{Q}$ to $\mathcal{Q}' = (M', E', root(\mathcal{Q}'))$ such that*

$$\begin{aligned} M' &= M \setminus \{v\}, \\ E' &= E \setminus \{(u, w) \mid (u, w) \in E \wedge w = v\}. \end{aligned}$$

Sequences of deleting inner nodes and leaves are not commutative. All deletions of inner nodes must precede all deletions of leaves. In the query depicted in Figure 3(a), only the leaf "piano" *or* the leaf "concerto" may be deleted (because a query leaf asking for the existence of a title does certainly not capture the intention of the user). However, after deleting the title node, the deletion of both the "piano" and the "concerto" leaf from the transformed query is permitted.

4.2.3. Node Renaming: Changing Context

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

Definition 7 (Renaming) *Let $\mathcal{Q} = (M, E, root(\mathcal{Q}))$ be a query tree and $v \in M$ be a node. A renaming of v is the change of its label from l_i to l_j by modifying the labeling function. The new function $label'$ is defined as follows:*

$$\forall u \in M : label'(u) = \begin{cases} l_j & \text{if} \quad u = v \\ label(u) & \text{else.} \end{cases}$$

4.3. The Approximate Query Matching Problem

In this subsection, we formally define the approximate query matching problem. We first define the embedding cost of a transformed query as the sum of the costs of all applied basic transformations.

Definition 8 (Embedding cost) *Let $\mathcal{Q}$ be a conjunctive query tree and $\mathcal{Q}'$ be a tree derived from $\mathcal{Q}$ by a sequence $t_1, t_2, \ldots, t_n$ of basic query transformations. Let $cost(t_i)$ be the cost of the insert, delete, or rename transformation represented by t_i. The embedding cost of $\mathcal{Q}'$ is defined by*

$$embcost(\mathcal{Q}') = \sum_{i=1}^{n} cost(t_i).$$

Note, that the embedding cost of a transformed query is independent from the data. To evaluate an approXQL query, the *closure* of transformed queries is created from the separated query representation, and all queries that have embeddings are selected. We define the notion of a *query closure*:

Definition 9 (Query closure) *Let $\mathcal{Q}$ be a query. The following formulas inductively create sets of transformed queries:*

$$\begin{aligned} \mathbb{Q}^0 &= \{\, \mathcal{Q}' \mid \mathcal{Q}' \text{ is element of the separated representation of } \mathcal{Q} \,\}, \\ \mathbb{Q}^{n+1} &= \{\, \mathcal{Q}' \mid \mathcal{Q}' \text{ is result of a basic transformation of } \mathcal{Q} \in \mathbb{Q}^n \,\}. \end{aligned}$$

The closure of $\mathcal{Q}$ is defined by

$$\mathbb{Q}^* = \bigcup_{n=0}^{\infty} \mathbb{Q}^n.$$

The closure of an approXQL query is a set of conjunctive query trees. Every query $\mathcal{Q} \in \mathbb{Q}^*$ is executed against $\mathcal{D}$. Executing a query means finding a (possibly empty) set of embeddings of $\mathcal{Q}$ in $\mathcal{D}$ according to Definition 1 at Page 6. All transformed queries are divided into *embedding groups*:

Definition 10 (Embedding group) *Let $\mathbb{Q}^*$ be the closure of a query and $\mathcal{D}$ be a data tree. An embedding group F_v is a set of embedding-cost pairs such that all embeddings have the same root v:*

$$F_v = \{\, (f, c) \mid \mathcal{Q} \in \mathbb{Q}^*, f \text{ is an embedding of } \mathcal{Q} \text{ in } \mathcal{D}, c = \mathit{embcost}(\mathcal{Q}), \text{ and } f(\mathit{root}(\mathcal{Q})) = v \,\}$$

We say that v is the *embedding root* of the group F_v. Each embedding group represents a result of the query. To rank the list of results, we need a single score for each result. Since several embeddings (collected in an embedding group) may select the same result, we choose the embedding with the smallest embedding cost. Putting the pieces together, we now ready to formalize the *approximate query matching problem*:

Definition 11 (Approximate query matching problem) *Given a data tree $\mathcal{D}$ and the closure $\mathbb{Q}^*$ of a query, locate all embedding groups. Represent each group F_v by a pair (v, c) consisting of the embedding root of F_v and the smallest embedding cost $c = \min\{\, c \mid (f, c) \in F_v \,\}$ of the group.*

The following steps summarize the evaluation of an approXQL query $\mathcal{Q}$ against the data tree $\mathcal{D}$:

1. Break up $\mathcal{Q}$ into its conjunctive normal form.
2. Derive the closure $\mathbb{Q}^*$ of transformed queries from $\mathcal{Q}$.
3. Try to exactly embed each query $\mathcal{Q} \in \mathbb{Q}^*$ into $\mathcal{D}$.
4. Divide the embeddings into embedding groups.
5. From each group, choose the embedding with the lowest cost.
6. Rank the embedding roots according to their cost.

These six steps describe the evaluation of an approXQL query from the theoretical point of view. Obviously, the brute-force generation of the closure is not a very smart way of query evaluation. In fact, many transformed queries can be discarded at an early stage using information about the data structure. In Section 6 we shall see a query-evaluation method that encodes all deletions and renamings in a *single* query tree and derives the cost of all insertions from a special encoding of the data tree.

Example: To illustrate the six steps of evaluating an approXQL query, we use the example query

$\mathcal{Q}$ = `cd[title["piano" $and$ "sonata"] $and$ performer["rachmaninov"]]`.

Note, that we assign the symbol $\mathcal{Q}$ to the textual representation of the query $\mathcal{Q} = (M, E, \mathit{root}(\mathcal{Q}))$ throughout the example. For simplicity, we consider only a very restricted set of query transformations:

basic transformation	cost
deleting "sonata"	8
renaming performer to composer	5
renaming "sonata" to "concerto"	3

All other basic transformations get the cost ∞. In particular, all insertions have infinite cost. In the following, we only consider transformed queries with embedding costs less than infinite.

In the first step, the separated query representation is generated. Since the $\mathcal{Q}$ does not contain `$or$` operators, the separated representation consists of a single conjunctive query tree only:

{ `cd[title["piano" $and$ "sonata"] $and$ performer["rachmaninov"]]` }.

In the second step, the sets of transformed queries are created:

$\mathbb{Q}^0$ = { `cd[title["piano" $and$ "sonata"] $and$ performer["rachmaninov"]]` },
$\mathbb{Q}^1$ = { `cd[title["piano"] $and$ performer["rachmaninov"]]`,
`cd[title["piano" $and$ "concerto"] $and$ performer["rachmaninov"]]`,
`cd[title["piano" $and$ "sonata"] $and$ composer["rachmaninov"]]` },
$\mathbb{Q}^2$ = { `cd[title["piano"] $and$ composer["rachmaninov"]]`,
`cd[title["piano" $and$ "concerto"] $and$ composer["rachmaninov"]]` }.

The sets are united to form the query closure $\mathbb{Q}^* = \mathbb{Q}^0 \cup \mathbb{Q}^1 \cup \mathbb{Q}^2$ of $\mathcal{Q}$. In the third step, the queries in $\mathbb{Q}^*$ are executed against the data tree shown in Figure 3 at Page 6. There is only one result in the depicted part of the tree – the subtree rooted at the node cd. All two transformed queries in $\mathbb{Q}^2$ can be embedded into the result. Since both embeddings belong to the same result, only one embedding group is generated in step four. According to our basic cost assignment, the query

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

from $\mathbb{Q}^2$ has the total transformation cost 13, because the keyword "sonata" has been deleted (cost 8) and performer has been renamed to composer (cost 5). The second query in $\mathbb{Q}^2$,

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

has the total transformation cost 8, because "sonata" has been renamed to "concerto" (cost 3) and performer has been renamed to composer (cost 5). In step five, the smallest embedding cost (8) of the group is chosen. The pair consisting of the embedding root and the embedding cost is retrieved to the user in step six.

4.4. Discussion: Basic Cost Assignment

So far, we simply assumed the existence of a cost for each basic query transformation. In this subsection, we take a closer look at the basic costs and the functions that assign the costs to nodes. Obviously, the cost of a basic query transformation should depend on the properties of the node(s) involved, which include the node type (*text* or *struct*), the node label, and the position of the node in the query tree or data tree, respectively. The several methods to assign costs we shall discuss in this subsection differ (1) in their ability to capture the semantics of nodes and inter-node relationships, and (2) in the effort needed to assign the costs. Let $S = M \cup N$ be the set of all nodes occurring in a query and in a data tree.

Node-specific costs. Every node $u \in S$ gets a dedicated insertion cost, and a deletion cost; every pair $u, v \in S$ of nodes gets a renaming cost.

Label-specific costs. This approach binds costs to node labels. Different nodes having the same label carry the same costs.

Path-specific costs. This variant tracks the different roles of a node label (e.g., CD title, MC title, track title). The cost of a node $u \in S$ with label l depends both on l and on the labels of the nodes along the path from the root node to n. This is exactly equivalent to the assignment of costs to the nodes of a strong DataGuide [GW97] belonging to the query tree or data tree, respectively.

DTD-specific costs. In a Document Type Description (DTD), every element name can appear only once, and every element can only have one content model. This content model describes the semantics of the element – independent of the place the element appears in an XML document that instantiates the DTD.

In principle, all of the above methods can be used in our framework. However, the assignment of node-specific costs is not practicable, since the definition of all rename costs for a given query would require the inspection of the entire data tree. For the rest of the paper, we assume *label-specific costs* as the simplest method of assigning costs.

5. Expressing User Preferences

The querying model introduced in the previous section uses query transformations in order to find results similar to the original query. The model is based on the assumption that the user prefers exact matches but is also interested in similar results. Sometimes, the user may not want to allow all possible transformations. For instance, the user may want to find only CDs that contain *all* the specified keywords in the CD title (and not in a track title). She may also want to find sound storage media that are in fact *composed* by Rachmaninov. In other situations, the user may want to assign the same score to results that have different structure. For instance, she may want to express that a query should treat CD titles and track titles equally: Both variants should get the same score and thus the same position in the ranking.

We support both types of user preferences in our model. Any *restriction* requested by the user forbids the corresponding query transformation; any desired *relaxation* removes the cost from the corresponding transformation. In addition, the user can assign a dedicated delete cost to each name and keyword of the query.

5.1. Query Restrictions

Restrictions forbid query transformations or impose additional limitations on the embedding functions. We support five types of restrictions concerning the insertion, deletion, and renaming of nodes, the order of nodes in the document, and the matching of strings.

An *insert restriction* forbids the insertion of nodes into a certain part of the query; a *rename restriction* does not allow to rename a query node; and a *delete restriction* suppresses the deletion of a node. Each restriction belonging to one of these three types is expressed by an exclamation mark, which is assigned to the name or keyword it is intended to affect. For instance, in the query

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

the nodes cd and "concerto" must not be renamed. The query

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

shows an insert restriction: It is not allowed to insert nodes between the nodes cd and title. Using this syntax, the parent-child selector "/" of XQL can be simulated. Finally, in the query

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

the keywords "piano" and "concerto" as well as the inner node composer must not be deleted. Note, that the deletion rule of vague containment (Definition 5 at Page 7) allows the deletion of inner query nodes bottom up. Thus, if the user forbids the deletion of an *inner* query node then the deletion of all its ancestors is forbidden implicitly. The three types of restrictions can be used in combination:

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

Ordering is the fourth type of restriction. The keyword `$followedby$` has the same semantics as the operator `$and$` but requires that the match of the first operand precedes the match of the second operand. The query

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

matches a cd subtree of the data tree in which the title subtree is on the left of the composer subtree, and the keyword "piano" precedes the keyword "concerto".

The `text()` operator of approXQL matches *substrings*. To enforce the selection of entire strings, the `content()` operator can be used as fifth type of restriction:

```
cd[title[content() = "piano concerto"] $and$
   composer[content() = "rachmaninov"]].
```

If the user forbids all query transformations then the set $\mathbb{Q}^*$ consists of the separated representation of the original query only. Since all queries in $\mathbb{Q}^*$ have embedding cost 0, only exact matches are retrieved. Thus, the semantics of a "fully restricted" approXQL query is equivalent to the corresponding XQL query.

5.2. Query Relaxations

Relaxations remove the cost from the respective transformations. We distinguish between *rename relaxations* and *insert relaxations*. The following example demonstrates insert relaxations:

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

The title node matches CD titles and CD track titles with no difference regarding the cost. Similarly, the a node with label composer may be appear at any distance to its ancestor node with label cd, and the keyword "rachmaninov" may appear at any depth in the composer part of the document. The relaxation operator "*" of approXQL is exactly equivalent to the XQL operator "//": It skips any number of nodes without imposing costs.

Rename relaxations allow the mapping of a query name to different document names without using a rename cost. In the query

```
(cd | mc)[title["piano" $and$ "concerto"] $and$
         (composer | performer)["rachmaninov"]].
```

The user specifies that CDs and MCs have to be treated equally. In the same way, the user wants to select works composed or performed by Rachmaninov without difference.

5.3. Assigning Delete Costs

The user typically does not know beforehand which nodes will be inserted into the query in order to find the best results. Similarly, she typically does not know which renamings are possible. However, she always knows which query nodes can be deleted. Therefore, it is useful to allow the user to affect the default delete cost of the query nodes. We use the syntax *label:cost* to provide a deletion cost of the query node having label *label*:

```
cd[title:2["piano" $and$ "concerto":5] $and$
   (composer | performer):3["rachmaninov"]].
```

In this query, the cost of deleting the title, composer or performer node is 2, and deleting the keyword "concerto" has the cost 5. Note, that the expression *label:!* introduced in Section 5.1 is a synonym for $label : \infty$.

If the user applies the sign "+" to the cost specification, then the he value is added to the default cost. Similarly, the sign "-" subtracts the value from the default cost. In the following example, the delete cost of the title is decreased by 2, whereas the delete cost of the keyword "piano" is increased by 4:

```
cd[title:-2["piano":+4 $and$ "concerto"].
```

The query interpreter ensures that the delete cost of a query element cannot be set to a value less than zero.

6. A Polynomial Query-Evaluation Algorithm

The approximate query matching model introduced in Section 4 explicitly creates an (infinite) set of transformed queries from a user-provided query. This model seems to require an implementation that has an exponential runtime complexity. Fortunately, as we shall see in this section, the approximate query matching problem can be solved by an algorithm that has polynomial – typically sublinear – worst-case time complexity.

We first investigate the evaluation of a *conjunctive* query tree: Using a special encoding of the data tree, we determine how many nodes must be inserted in the query in order to find exact embeddings. At this stage, no deletions and renamings of query nodes are permitted.

In a second step, we introduce the *Boolean representation* of a user query, which represents the approXQL operators `$and$` and `$or$` as special nodes, avoiding the explicit creation of the separated query representation.

In the third step, we transform the Boolean query representation to an *expanded query representation*, which encodes all allowed deletions and renamings of query nodes in a single tree. By adapting the evaluation principle for conjunctive queries to the expanded representation of a user query, we are able to find all approximate embeddings of the query in a single-pass iteration through the expanded query tree.

6.1. Encoding of the Data Tree, Partial Index, and Postings

The embedding of a conjunctive 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 tree. This "blind" insertion of nodes creates many queries that have no embedding at all. We completely avoid the insertion of nodes into a query tree. Instead, we use a special encoding of the data tree in order to determine the *distance* between the matches of two query nodes. More precisely, we change property 4 of the embedding function (see Definition 1 at Page 6) from "parent-child preserving" to "ancestor-descendant preserving". For each parent-child pair of the query, the index retrieves all matching ancestor-descendant pairs together with the distance. The distance between two nodes u and v is the sum of the insert costs of all nodes along the path from u to v (excluding u and v).

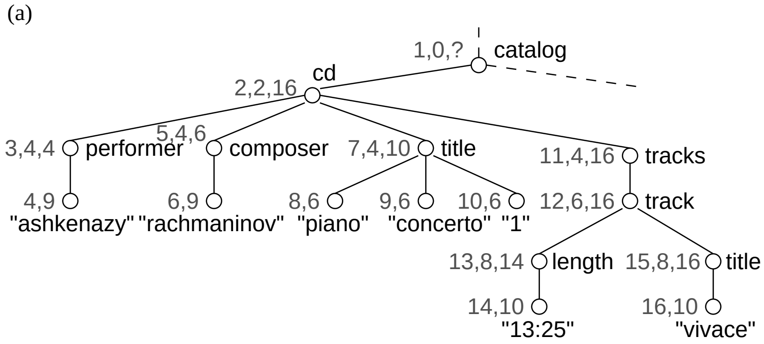
We use an indexing technique that is inspired by the *partial index* introduced in [Nav95]. A partial index consists of text index and a structural index. The text index maps every keyword to a list of text positions the keyword occurs at. The structural index maps every element name to all its occurrences in the document collection using two lists: The first one stores the start positions of text segments covered by the node. The second list indicates the corresponding final positions.

Our approach differs from [Nav95] since we (1) do not rely on text positions but on a preorder numeration of the data tree, (2) allow for recursive labeling, that is, the same label may occur twice or more on a document path, and (3) store additional distance information. Every structural node u is represented by a triple of integer values:

- $pre(u)$ is the preorder number of u,
- $dist(u)$ is the sum of the insert costs of all ancestors of u, and
- $bound(u)$ is the preorder number of the rightmost leaf of the subtree rooted at u.

Since the preorder number and the bound value of a text node are equal, we can omit the bound value for text nodes. An example of an encoded data tree is shown in Figure 4(a). We use the notation $u = (pre(u), dist(u))$ to refer to the encoding of a text node u, and $u = (pre(u), dist(u), bound(u))$ to denote the encoding of a structural node. Given two nodes u and v we can test if u is an ancestor of v by ensuring the invariant

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



(b)

1	: (10,6)
13:25	: (14,10)
ashkenazy	: (4,9)
concerto	: (9,6)
piano	: (8,6)
rachmaninov	: (6,9)
vivace	: (16,10)

(c)

catalog	: (1,0,?)
cd	: (2,2,16)
composer	: (5,4,6)
length	: (13,8,14)
performer	: (3,4,4)
title	: (7,4,10),(15,8,16)
track	: (12,6,16)
tracks	: (11,4,16)

Figure 4: Encoded XML tree (a) with its text index (b) and structural index (c).

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

$$dist(v) - dist(u) - c_{ins}.$$

The indexes map node labels to *postings*. In a text index, a posting P belonging to keyword l is a list of $(pre, dist)$ pairs representing all occurrences of l in the data tree. Similarly, each posting P belonging to label l of a structural index is a list of $(pre, dist, bound)$ tuples representing all occurrences of l. All postings are sorted by the preorder node numbers ascendingly. We often use the shorthand term "node" to denote a tuple representing a node. The number of entries of a posting is denoted by $|P|$; with the notation P_i we refer to the ith tuple of the posting P. The expression $P_{[i,j]}$ selects the interval starting at position i and ending at position j of P. An empty posting is denoted by [ ].

In the following subsections, we will frequently use the term *generated posting*. A generated posting is an intermediate results of our query answering algorithm. Each entry u of a generated posting is a tuple consisting of five values:

$$u = (pre(u), dist(u), bound(u), rencost(u), embcost(u)).$$

Entries of generated postings are initialized with entries of postings fetched from the index. The values of the components $pre(u)$ and $dist(u)$ are copied from the corresponding components. If the posting is fetched from the structural index, then the value of $bound(u)$ is copied from the entry. Otherwise, if the posting is fetched from the text index, the $bound(u)$ value is set to the value of the component $pre(u)$. The two additional components have the following semantics:

- $rencost(u)$ represents the renaming cost of the query node that is mapped to u. This value is only used in the "final" version of the query-evaluation algorithm described in Section 6.4.
- $embcost(u)$ stores the cost of embedding a query subtree into the data subtree rooted at u. The value is zero if u is the match of a query leaf.

We shall see in the next subsections how generated postings are used to compute all exact and approximate matches of a query.

6.2. Step 1: Conjunctive Query Trees

The evaluation algorithm for conjunctive query trees is based on the *dynamic programming principle*: The embedding cost of a query subtree rooted at a node u is calculated from

1. the embedding costs of the subtrees rooted at the children of u, and
2. the distances between the match of u and the matches of the children of u.

All matches of a query node with a given label are stored in a single posting. To find all embeddings of a query tree, the algorithm uses *operations on postings*. For conjunctive query trees, two types of operations are necessary: First, given a posting of potential ancestors and a posting of potential descendants, the algorithm must find all ancestor-descendant pairs with minimal distance. Second, given two postings that represent the embeddings of a query node with respect to different children, the algorithm must find all pairs that belong to the same data node. From each of these pairs, the sum of the embedding costs must be calculated. Both types of operations are described in the following subsections.

6.2.1. Joining the Ancestor-Descendant Relationship: The Function `join_path`

The function `join_path` takes a posting P of potential ancestors and a posting R of potential descendants. For each node of P the function searches in R for the descendant with the smallest embedding cost. Let P_i be an ancestor, R_j be a descendant, and c_{ins} be the insert cost of the node represented by P_i. Then, the embedding cost of P_i is defined as

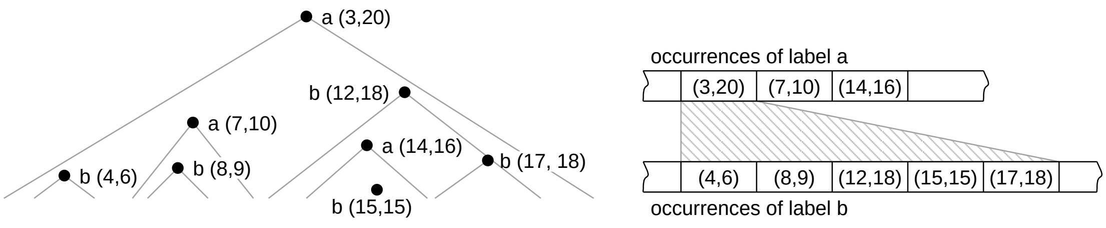
$$embcost(P_i) = embcost(R_j) + dist(R_j) - dist(P_i) - c_{ins},$$

where the last three terms represent the formula to calculate the distance between two nodes (see Section 6.1). Note, that the function requires that the embedding cost of the descendants has already been calculated.

Recall that the postings are sorted by the preorder node numbers ascendingly. If the data tree is not recursive (that is, no node has an ancestor or descendant with the same label), then the ancestor-descendant relationship can be established by simultaneously iterating through the postings P and R [Nav95]. In particular, all n descendants of a node P_i reside in a interval $R_{[j,j+n]}$. The smallest embedding cost of P_i with respect to descendants in R is calculated using the formula

$$embcost(P_i) = \min\{\, embcost(R_k) + dist(R_k) \mid j \le k \le j+n \,\} - dist(P_i) - c_{ins}.$$

However, XML documents may have a recursive structure which we cannot ignore. The linear join does not work for recursive structures since some intervals encoded in the postings may overlap. Fortunately, we can adopt the join algorithm such that it can cope with recursive trees but retains its linear behavior in the case of non-recursive trees. This modification relies on a simple observation concerning the positions of the descendants of a node in a *single* posting: Let P_i be a node in P. If P_i has a descendant that has the same label as P_i, then this descendant must be in P, too. Moreover, because the data tree is preorder enumerated, and because P is sorted, all m descendants of P_i in P must reside at the interval $P_{[i+1,i+m]}$. We now consider the positions of descendants in two *different* postings: Assume that we have already found the interval $R_{[j,j+n]}$ of descendants of P_i. If the nodes in the interval $P_{[i+1,i+m]}$ have descendants in R, then



(a) Part of a recursive data tree.

(b) Postings with nested intervals.

Figure 5: A part of recursive data tree and the corresponding sections from the postings belonging to the labels a and b, respectively. The distance values are not shown.

those descendants must reside in the interval $R_{[j,j+n]}$, too. Figure 5 illustrates this observation: All b-labeled descendants of node 3 reside in a contiguous interval of the posting that stores all occurrences of b. If we already know that interval then we also know that all a-labeled descendants of node 3 (nodes 7 and 14) have its b-labeled descendants in the same interval as node 3. We can restrict the search for descendants to the computed intervals if we observe recursive labeling. The function `join_path` uses recursive calls to realize the interval search.

Algorithm 1 joins the ancestor-descendant relationship for recursive trees.

function `join_path`(P, R, c_{ins}, c_{del})
1: $S := [\,]$; $i := 0$; $j := 0$
2: **while** $i < |P|$ **do**
3:     **while** $j < |R|$ **and** $pre(R_j) \le pre(P_i)$ **do**
4:         $j := j + 1$
5:     $c_{min} := \infty$; $j' := j$
6:     **while** $j < |R|$ **and** $pre(R_j) \le bound(P_i)$ **do**
7:         $c_{min} := \min(c_{min}, embcost(R_j) + dist(R_j))$
8:         $j := j + 1$
9:     $c_{min} := \min(c_{del}, c_{min} - dist(P_i) - c_{ins})$
10:     **if** $c_{min} < \infty$ **then**
11:         Append S by a copy of P_i and set its embedding cost to c_{min}
12:     $i := i + 1$; $i' := i$
13:     **while** $i < |P|$ **and** $pre(P_i) \le bound(P_{i'})$ **do**
14:         $i := i + 1$
15:     **if** $i' < i$ **and** $j' < j$ **then**
16:         Append S by the results of the call `join_path`$(P_{[i',i]}, R_{[j',j]}, c_{ins}, c_{del})$
17: **return** S

Algorithm 1 shows the function `join_path`. It works as follows: First, the posting S that will contain the results of the join is initialized by the empty list (Line 1). At Lines 3 and 4, the position of an interval of descendants in R is searched. Among all descendants in that interval, the one with the smallest sum of distance and embedding cost is chosen (lines 6-8). The embedding cost of P_i is the minimum of this sum and the delete cost of the corresponding query node passed as parameter c_{del} (Line 9). Only of the embedding cost is less than infinite, a copy of P_i is appended to S (Lines 10 and 11). At Lines 13-16, recursive structures are taken into account. The algorithm searches the first node in P that is *not* a descendant of P_i (Lines 13 and 14). If the node P_i does not have any descendants in R, then all its descendants in P cannot have descendants in R as well. Otherwise, if P_i has both descendants in P and in R, then the algorithm calls the function recursively for the computed intervals in P and R (Line 16). The parameter c_{del} will be needed by the query-evaluation algorithm for expanded queries only; it is ∞ if the function is used in algorithms that do not allow node deletions.

6.2.2. Summary of two Postings: The Function `compute_sum`

The function `compute_sum` creates a new posting that consists of all nodes that occur in both postings P and R, which are passed as parameters. The embedding cost of a generated node is the total cost of the embedding costs of its operands plus the cost c_{del}. Algorithm 2 shows the function. The parameter c_{del} is greater than zero only if the deletion of nodes is permitted.

6.2.3. Evaluating a Conjunctive Query Tree

The evaluation algorithm for conjunctive query trees computes the embedding cost of a query node using the embedding costs of its children. Starting at the root of the conjunctive query tree, the algorithm visits the nodes in depth-first order. During the descent it fetches from the index

Algorithm 2 creates a posting that consists of nodes occurring in both operands.

function `compute_sum`(P, R, c_{del})
$S := [\,]$; $i := 0$; $j := 0$
while $i < |P|$ **and** $j < |R|$ **do**
  if $pre(P_i) = pre(P_j)$ **then**
    Append S by a copy of P_i and set its emb. cost to $embcost(P_i) + embcost(R_j) + c_{del}$
  if $pre(P_i) \leq pre(P_j)$ **then**
    $i := i + 1$
  if $pre(P_i) \geq pre(P_j)$ **then**
    $j := j + 1$
return S

the postings belonging to the labels of the nodes. Arrived at the leftmost leaf, it joins the posting belonging to this leaf with the posting belonging to the parent of the leaf. If a node u has two or more children, then the cheapest combination of matches belonging to u's children is chosen and stored in the posting generated for u. The posting returned by the algorithm contains all embedding roots of the query together with the embedding costs.

Algorithm 3 shows the query-evaluation procedure. Let u be the root of a conjunctive query tree and [ ] be an empty posting. Then the following statement triggers the evaluation of the query:

$$P := \text{evaluate}(u, [\,]).$$

The results are stored in the posting P. Note, that parameter c_{del} of the functions `join_path` and `compute_sum` are initialized by the values 0 and ∞, respectively since no deletions are allowed at this stage.

Algorithm 3 evaluates a conjunctive query tree.

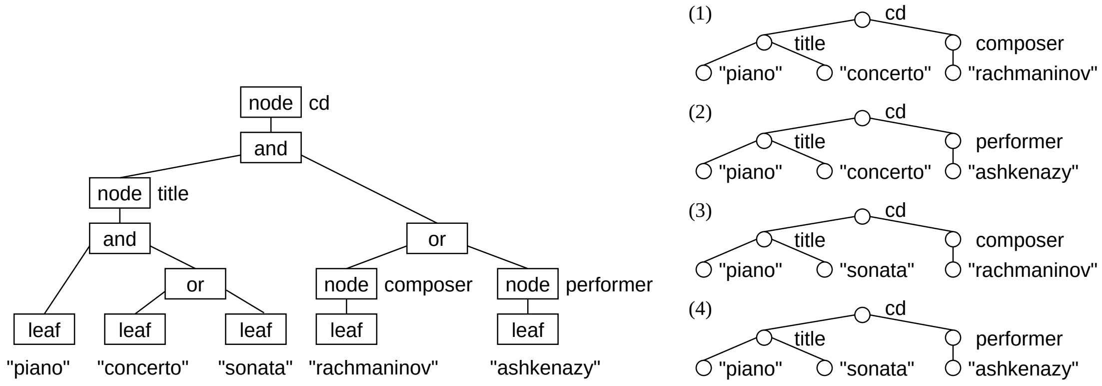
function evaluate(u, P)
Retrieve the posting R belonging to $label(u)$ from the text/structural index
if $|children(u)| > 0$ **then**
  $R :=$ `evaluate`$(child_1(u), R)$
  for $i := 2$ **to** $|children(u)|$ **do**
    $S :=$ `evaluate`$(child_i(u), R)$
    $R :=$ `compute_sum`$(R, S, 0)$
return `join_path`$(P, R, inscost(u), \infty)$

6.3. Step 2: Boolean-Connected Query Trees

In the previous subsection we have seen that all embeddings of a *conjunctive* query tree can be found by dynamic programming. In this subsection, we shall see that the explicit creation of the separated representation of a query is not necessary: Because the queries in the decomposed representation share common subtrees, and because the embedding costs calculated for different query branches do not influence each other, the entire approXQL query can be represented by a single *Boolean query representation.*

6.3.1. The Boolean Query Representation

The Boolean representation of an approXQL query is a tree consisting of four node types: A node of type *node* represents an inner name selector of the user query, that is, a name selector that is followed by a containment operator. All name selectors without a following containment operator



(a) Boolean representation (b) Separated representation

Figure 6: Boolean representation versus separated representation of the query
`cd[title["piano" $and$ ("concerto" $or$ "sonata")] $and$ (composer["rachmaninov"] $or$ performer["ashkenazy"])]`

as well as all text selectors are represented as nodes of type *leaf*. Logical connections between the children of a name selector are mapped to a binary tree consisting of *and* and *or* nodes. Figure 6(a) shows a Boolean query representation. The separated representation of the same query is depicted in Figure 6(b).

We define the following functions for every node u in the Boolean query representation: $type(u)$ returns the type of u, which can be either *and*, *or*, *node*, or *leaf*. For every node of type *node*, the function $inscost(u)$ returns the cost of inserting the node u into a query. This function is used to compute the distance between an ancestor-descendant pair (see Section 6.1).

6.3.2. Minimum of two Postings: The Function `compute_min`

The function `compute_min` is executed at each *or* node of the Boolean query representation. For each embedding root represented by two entries P_i and R_j, it selects the entry with the cheapest embedding cost and takes the cost in the corresponding entry of the generated posting.

Algorithm 4 creates a posting that consists of nodes occurring in either operand.

function `compute_min`(P, R, c_{del})
$S := [\,]$; $i := 0$; $j := 0$
while $i < |P|$ **or** $j < |R|$ **do**
  if $j = |R|$ **or** $(i < |P|$ **and** $pre(P_i) < pre(R_j))$ **then**
    Append S by a copy of P_i and increment its embedding cost by c_{del}
    $i := i + 1$
  else if $i = |P|$ **or** $(j < |R|$ **and** $pre(P_i) > pre(R_j))$ **then**
    Append S by a copy of R_j and increment its embedding cost by c_{del}
    $j := j + 1$
  else /* $pre(P_i) = pre(R_j)$ */
    Append a copy of P_i to S and set its emb. cost to $\min(embcost(P_i), embcost(R_j)) + c_{del}$
    $i := i + 1$; $j := j + 1$
return S

6.3.3. Evaluating a Boolean Query Representation

The Boolean query representation is used as input for an algorithm that computes all embeddings of the query – without the explicit generation of the separated representation of the query. Instead of embedding each query of the separated representation independently and rejecting the "bad" queries afterwards, the algorithm decides locally at the *or* nodes of the Boolean query representation, which query branch has to be chosen. Each choice at an *or* node discards one or more queries of the separated query representation.

As an example, consider the title node of the query shown in Figure 6. If the left branch of the *or* connecting the leaves "concerto" and "sonata" has a smaller embedding cost (with respect to a certain match of the title node) than the right branch, then only the smaller embedding cost has to be tracked. This is equivalent to the removal of the queries 3 and 4 from the separated query representation. Similarly, if the branch composer["rachmaninov"] has a smaller embedding cost than the branch performer["ashkenazy"] then this is equivalent to the removal of the queries 2 and 4 from the separated query representation. In summary, choosing a branch of an *or* node is equivalent to removing all queries from the separated representation that do not have this branch.

Algorithm 5 extends Algorithm 3 to work with Boolean query representations. It makes use of the function `compute_min` to select the cheapest branch of every *or* node.

Algorithm 5 evaluates a query based on its Boolean representation.

---

function evaluate(q, P)
  case $type(q)$ **of**
    leaf: Retrieve the posting R belonging to $label(q)$ from the text/structural index
        $P :=$ `join_path`$(P, R, inscost(q), delcost(q))$
    node: Retrieve the posting R belonging to $label(q)$ from the structural index
        $R :=$ `evaluate`$(child_1(q), R)$
        $P :=$ `join_path`$(P, R, inscost(q), \infty)$
    and: $P :=$ `compute_sum`(`evaluate`$(child_1(q), P)$, `evaluate`$(child_2(q), P)$, $sumdelcost(q)$)
    or: $P :=$ `compute_min`(`evaluate`$(child_1(q), P)$, `evaluate`$(child_2(q), P)$, $sumdelcost(q)$)
  return P

6.4. Step 3: Expanded Query Representations

The expanded representation of a query extends its Boolean representation. It explicitly represents all deletions and renamings of query nodes that have a cost less than infinite. Using the expanded query representation, the dynamic-programming principle can be adapted in order to find all *approximate* embeddings of Boolean queries. In Section 4 we stated that the number of transformed queries may be infinite. Therefore, it might be surprising that all renamings, deletions, and even the logical operators can be represented in a single tree. Note, however, that the insert operation is the only operation that may lead to an infinite number of transformed queries. In Section 6.2, we demonstrated that the explicit insertion of nodes in a query is not necessary because the distance between nodes can by determined from the encoded data tree. We applied the dynamic-programming principle to Boolean queries in Section 6.3. Now we shall discuss that even the representation of all node deletions and renamings in the query tree does not lead to a combinatorial explosion.

6.4.1. The Expanded Query Representation

The *expanded representation* of a query is based on the Boolean tree introduced in Section 6.3. It extends the Boolean representation by encoding all permitted deletions and renamings of nodes explicitly.

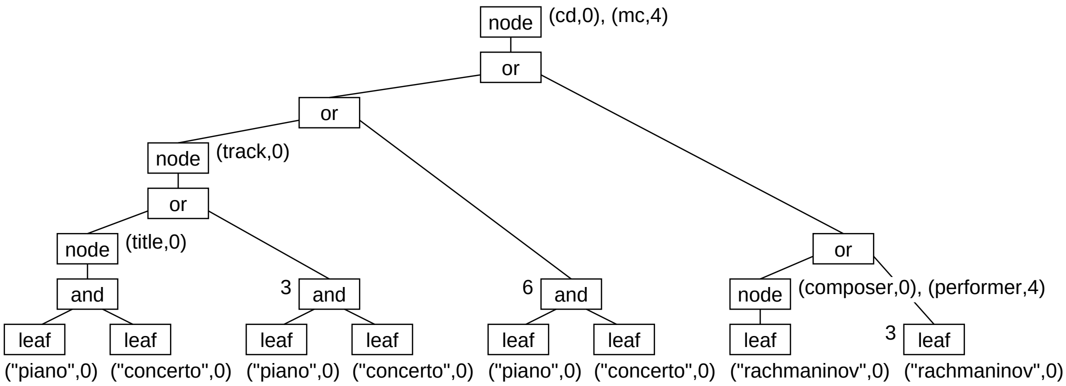


Figure 7: Expanded representation of the query
`cd[track[title["piano" $and$ "concerto"]] $or$ composer["rachmaninov"]]`

Node deletions. All permitted deletions of inner query nodes are represented explicitly in the tree: Every inner query node (except the root) is replaced by a binary *or* node. The left child refers to the query node; the right child represents the fact that the query node is deleted.

Figure 7 shows an expanded query representation. The only inner nodes that may be deleted are title, composer, and track (after title has been deleted). They are represented by *or* nodes. For example, the left-most *or* node in Figure 7 represents the deletion of the title selector of the user query. The left child of the *or* node refers to a tree that models the subquery title["piano" and "concerto"]. The right child of the *or* node is a subtree that represents the and connection of the keywords "piano" and "concerto" in the context track.

For each node u that has either type *node* or type *leaf*, a function $delcost(u)$ is defined. This function returns the cost of deleting u from the query. Recall, that for each node of type *node*, a function $inscost(u)$ is defined as well. In addition, we define a function $sumdelcost(u)$ for every node. The function returns the summary cost of deleting the query node and all its inner descendants. The returned value is greater than zero only for the right child of an *or* node that represents the deletion of an inner node. In our example query depicted in Figure 7, all delete transformations have the uniform cost 3. Therefore, the summary delete cost of the second left *and* node is 3 because it represents the deletion of the title node. Similarly, the summary delete cost of the third left *and* node is 6 because it represents the deletion of both the track node and the title node. All non-zero summary delete costs of our example query are shown at the left-upper corner of the node boxes in Figure 7.

Node renamings. Nodes that have either the type *node* or the type *leaf* are annotated with a set of pairs, each storing a label and a value. A label l represents a possible renaming of the node; the associated value represents the cost to replace the original node label by l In our example, the costs of renaming cd into mc and composer into performer are both 4. The set label-cost pairs belonging to a node u are retrieved by a function $labels(u)$.

6.4.2. Index Access and Merging of Postings: The Function `merge`

The index access of the previous variants of our query-evaluation algorithm was simple, because every query node had only one label. The task of the function `merge` is to fetch and merge all postings that belong to any of the labels in the label-cost set of a node. For each label in this set, the function accesses the index, gets the posting, and merges it with the posting belonging to all other labels in the passed set. The posting retrieved by the function `merge` is sorted by the preorder numbers its nodes. Algorithm 6 shows the function.

Algorithm 6 fetches and merges the postings belonging to the label set of a query node.

function `merge`(L, c_{del})
$S := [\,]$
for each $(l, c_{ren}) \in L$ **do**
  Retrieve the posting P belonging to label l from the text/structural index
  for $i := 0$ **to** $|P| - 1$ **do**
    Insert a copy of P_i into S such that S remains sorted
    Set the renaming cost of the copy to c_{ren} and its embedding cost to c_{del}
return S

6.4.3. Adding the Rename Cost: The Function `add_rencost`

The function `add_rencost` takes a posting P and iterates through P. For each entry of P, the function adds the value of the *rencost* component to the embedding cost of the same entry. Note, that the only purpose of this function is to simplify the explanation of the query-evaluation algorithm. In practice, the functionality of this function is integrated in the functions `compute_sum` and `compute_min`.

Algorithm 7 adds the rename cost to the embedding cost for every node in the posting.

function `add_rencost`(P)
for $i := 0$ **to** $|P| - 1$ **do**
  Add $rencost(P_i)$ to the embedding cost of P_i
return P

6.4.4. Evaluating an Expanded Query Representation

We are now ready to present the final version of the query-evaluation algorithm, which finds all approximate query embeddings using the expanded representation of a query and the functions introduced in the previous subsections.

Algorithm 8 starts at the query root and visits the nodes of the expanded query tree in depth-first order. For each node of type *node* the algorithm passes, a posting consisting of all matching data nodes is constructed using the function `merge`. Then, the costs of the cheapest embeddings are computed for every subtree of the expanded query using the functions `join_path`, `compute_sum`, `compute_min`, and `add_rencost`. Finally, the algorithm returns a generated posting P that contains the root nodes of all results together with the embedding costs.

Note, that the algorithm can be easily modified to track not only the embedding costs but also the matching nodes. In this way, the best matching query can be generated for each result. Algorithm 8 is slightly simplified. It allows the deletion of all children of an inner query node – which is forbidden by Definition 6 at Page 8. To keep at least one child, the "full version" of the algorithm maintains an additional *delta* value per posting entry. This delta value represents the difference between the cheapest embedding and the cheapest embedding under the constraint that at least one child is not deleted. If all children of a query node of type *node* are processed, the delta value is added to the embedding cost.

6.5. Example

Consider the execution of the query

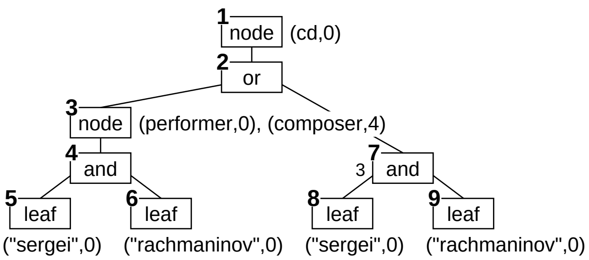
```
cd[performer["sergei" $and$ "rachmaninov"]]
```

against the data tree shown in Figure 4(a). First, the expanded representation of the query, depicted in Figure 8(a), is generated. To refer to the query nodes in this figure without ambiguity,

Algorithm 8 evaluates a query based on its expanded representation.

function evaluate(u, P)
1: **case** *type*(u) **of**
2: *leaf*: $R :=$ `merge`$(labels(u),\ sumdelcost(u))$
3: $R :=$ `add_rencost`(R)
4: $P :=$ `join_path`$(P, R, inscost(u), delcost(u))$
5: *node*: $R :=$ `merge`$(labels(u),\ 0)$
6: $R :=$ `evaluate`$(child_1(u), R)$
7: $R :=$ `add_rencost`(R)
8: $P :=$ `join_path`$(P, R, inscost(u), \infty)$
9: *and*: $P :=$ `compute_sum`(`evaluate`$(child_1(u), P)$, `evaluate`$(child_2(u), P)$, $sumdelcost(u)$)
10 *or*: $P :=$ `compute_min`(`evaluate`$(child_1(u), P)$, `evaluate`$(child_2(u), P)$, $sumdelcost(u)$)
11: **return** P

every node in this figure has a unique number, assigned to the upper-left corner of the node box. All insert, delete, and rename costs less than infinite are shown in Table 8(b). Node **7** has the summary delete cost 3, which represents the deletion of the performer node. All other summary delete costs are zero. For simplicity, we will use the notation $P^{\mathbf{x}}$ to refer to the posting belonging to node with number **x**, avoiding the context-dependent variables P and R of the algorithm.



(a) Expanded query representation

basic transformation	cost
inserting cd	2
inserting composer	5
inserting performer	5
inserting title	2
deleting performer	3
deleting "sergei"	5
deleting "rachmaninov"	5
renaming performer to composer	4

(b) Basic costs

Figure 8: Expanded representation and basic costs of the query
`cd[performer["sergei" $and$ "rachmaninov"]]`

The evaluation of the query starts with a call of the function `evaluate` (Algorithm 8), passing the query root and an empty posting as parameters. Because the query root is of type *node*, Line 5 of the algorithm it executed. The function `merge` takes the set of label-cost pairs $\{(\mathsf{cd}, 0)\}$, accesses structural index shown in Figure 4(c), and copies the posting $[\,(2, 2, 16)\,]$ into the generated posting $P^{\mathbf{1}} = [\,(2, 2, 16, 0, 0)\,]$. Algorithm 8 continues to Line 6 and calls the function `evaluate` recursively, passing both the child node **2** of the query root and the generated posting $P^{\mathbf{1}}$. The type *or* of node **2** triggers the execution of Line 10 of the algorithm. The function `compute_min` is applied to the resulting postings of two recursive calls to the function `evaluate`: one call for the left child (number **3**) of the *and* node, one call for its right child (number **7**). In both calls of the function `evaluate`, $P^{\mathbf{1}}$ is passed as parameter. We first look at the evaluation of the left child of node number **2**. At Line 5 of the algorithm, the postings belonging to the labels performer and composer, respectively, are fetched and merged into the posting $P^{\mathbf{3}} = [\,(3, 4, 4, 0, 0), (5, 4, 6, 4, 0)\,]$. This posting is passed to the function `evaluate` (Line 6), which continues the execution at the child node **4**. Arrived at this *and* node, its left child node labeled with "sergei" is evaluated. Because the type of this node is *leaf*, Line 2 of the algorithm is executed. No matches exist for the keyword "sergei", therefore, the posting $P^{\mathbf{5}}$ is empty. The function `join_path` at Line 4 joins the postings $P^{\mathbf{3}}$ and $P^{\mathbf{5}}$ and returns the posting $[\,(3,4,4,0,5),(5,4,6,4,5)\,]$. The embedding cost of both entries is 5 reflecting the deletion of the keyword "sergei". The same procedure is applied to the right child (labeled with "rachmaninov") of the node with number **4**: The posting $[\,(6,6)\,]$ is fetched from the index, merged into the generated posting $[\,(6,6,6,0,0)\,]$, and joined with the posting $P^{\mathbf{3}}$. The result of this operation is the posting $[\,(3,4,4,0,3),(5,4,6,4,0)\,]$. Now, the function `compute_sum` (Line 9) is applied to the results belonging to the children of node **4**. The result of the call to `compute_sum`($[\,(3,4,4,0,5),(5,4,6,4,5)\,],[\,(3,4,4,0,5),(5,4,6,4,0)\,],0$) is $P^{\mathbf{4}} = [\,(5,4,6,4,5)\,]$. The first entry has been deleted since neither "sergei" nor "rachmaninov" is contained in performer. In the next step, $P^{\mathbf{4}}$ is returned to the node **3**. Now, $P^{\mathbf{3}}$ is equal to $P^{\mathbf{4}}$. Next, function `add_rencost` is executed at Line 7 yielding the result $P^{\mathbf{3}} = [\,(5,4,6,4,9)\,]$. At Line 8, $P^{\mathbf{3}}$ is joined with $P^{\mathbf{1}}$. The result $[\,(2,2,16,4,9)\,]$ of this join is returned to node **2**. The algorithm now evaluates the right child of the node **2**, which represents the deletion of the performer node: It joins the posting $P^{\mathbf{1}}$ with the generated postings $P^{\mathbf{8}} = [\ ]$ and $P^{\mathbf{9}} = [\,(6,6,6,0,0)\,]$ belonging to the keywords "sergei" and "rachmaninov", respectively. The results of both joins are collected using the function `compute_sum` at Line 9. The embedding cost 13 of the single entry of the resulting posting $P^{\mathbf{7}} = [\,(2,2,16,4,13)\,]$ is the sum of the delete cost 5 of node **8**, the summary delete cost 3 assigned to node **7**, and the cost 5 of inserting a composer node into the query. The *or* node with number **2** passes the postings $P^{\mathbf{3}}$ and $P^{\mathbf{7}}$ to the function `compute_min`. The result of this call is the posting $P^{\mathbf{2}} = [\,(2,2,16,4,9)\,]$ because the smaller embedding cost of the left child is taken. Now, posting $P^{\mathbf{2}}$ is returned to node **1** such that $P^{\mathbf{1}} = P^{\mathbf{2}}$. Finally, the rename cost 0 is added to every node of $P^{\mathbf{1}}$ yielding the result $P^{\mathbf{1}} = [\,(2,2,16,4,9)\,]$. For the representation of the results, only the first and last component of each entry (representing the node number and the embedding cost, respectively), are of interest. Thus, the single-entry list $[\,(2,9)\,]$ is retrieved to the user.

6.6. Time Complexity

All functions used by Algorithm 8 have a linear time complexity in the number of nodes in the data tree. The following table lists the upper bound for every function.

`merge`	`join_path`	`compute_sum`	`compute_min`	`add_rencost`
$O(r \cdot s)$	$O(r \cdot s)$ / $O(r \cdot s \cdot h)$	$O(r \cdot s)$	$O(r \cdot s)$	$O(r \cdot s)$

In these formulas, r stands for the maximal number of permitted node renamings per query node. The letter s denotes the selectivity, that is, the maximal number of data nodes carrying the same label. Note, that s is equal to the number of entries of the largest posting. The right formula in the `join_path` column represents the time complexity of the function `join_path` for recursive trees. A tree is recursive if a label occurs twice or more along a path. The letter h denotes the height of the data tree, which restricts the number of equal-labeled nodes along a path.

Algorithm 8 calls the functions `merge`, `join_path`, and `add_rencost` for every "labeled" node of the expanded query representation. All nodes of type *node* or type *leaf* are labeled. The algorithm used the functions `compute_sum` and `compute_min` for every node of type *and* or *or*, respectively. Let n_l be the number of labeled nodes, and n_b be the number of Boolean nodes in an expanded query representation. Then, the time complexity of the query-evaluation algorithm is bound by

$$O(n_l \cdot (i + r \cdot s \cdot h) + n_b \cdot r \cdot s).$$

Note that n_l, n_b and r are small numbers in practical cases, and that every posting typically represents only a small fraction of the data nodes. Therefore, the expected runtime complexity of our algorithm is sublinear with respect to the database size. First experiments with our prototypical implementation indicate the correctness of our assumption.

7. The ApproXDB Prototype

In this section, we describe our prototypical implementation of an approximate query-answering system called approXDB. The approXDB system consists of six main components: The database kernel, the document loader, the indexer, the approXQL query processor, the abstract generator, and the graphical query editor. Figure 9 shows the architecture of the We describe the server-side modules in the following subsection, and introduce the query editor in Section 7.2. approXDB system.

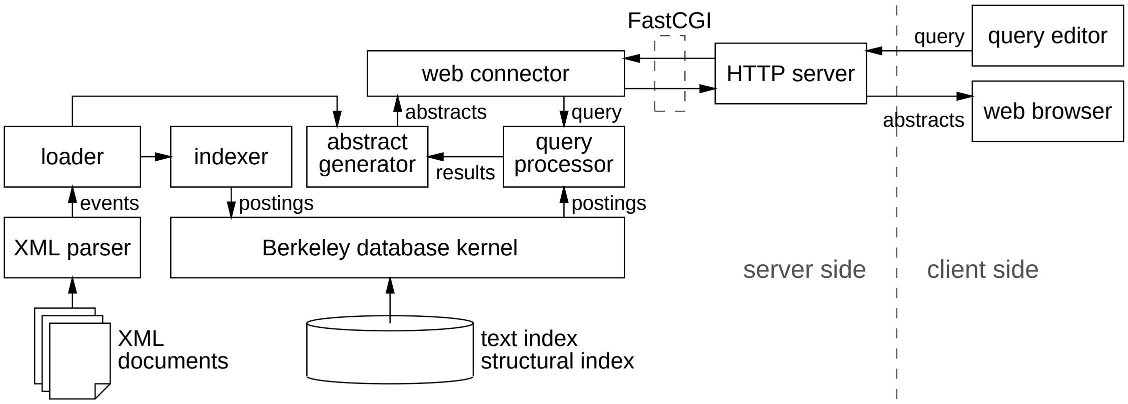


Figure 9: Architecture of the approXDB system.

7.1. The Server-Side Modules

The *kernel* is implemented on top of the Berkeley Database toolkit [BER00], which provides basic access structures as well as several subsystems. Both the structural and the text index are implemented as B^* trees. The leaves of the B^* trees store the postings, which are transfered into a memory area managed by the caching subsystem of the Berkeley database. The LRU (least-recently-used) strategy provided by the cache ensures that only postings that are frequently used reside in the cache; all other postings remain on disc. By analyzing the logs and by varying the cache size, the kernel can be adapted such that the ratio of cache misses is minimal given a fixed memory size.

On top of the kernel module reside the *loader* and the *indexer*. The loader uses the Xerces XML parser [XER00] to import XML files. It implements the SAX [Meg98] interface to the parser. The SAX interface provides callback methods that are triggered by the parser if a certain XML content type has been read. Whenever an opening tag of an element has been parsed, the posting belonging to the label is identified and the pair consisting of the element's preorder number and its distance to the root is added to the end of the posting. At this point, the *bound* value of the element is not yet known. Therefore, a pointer to the posting entry is pushed onto a stack. If the ending tag of the element has been read, the *bound* value is added to entry, and the pointer is removed from the stack. The loader splits text sequences into keywords and stems them. Keywords, attribute values, attribute names, and empty elements are added to the index immediately. Note, that – despite to the use of an additional stack – the time to construct the index is linear in the number of elements, attributes, and keywords in a document. The loader additionally updates an index that maps the pair consisting of preorder number and bound value belonging to the document's root to URL of the document. Given a node number $pre(u)$, the URL of the document containing the node can be found by searching the pair $(pre(v), bound(v))$ for which $pre(v) \leq pre(u) \leq bound(v)$ hold. We implemented this approach by simply replacing the comparison function of a B^* tree.

The *query processor* realizes the query-evaluation algorithm presented in Section 6. However, the implemented version of the algorithm is optimized: It does not iterate through the query tree recursively. Instead, a postorder numeration is applied to all nodes of types *node* and *leaf* – but not to Boolean nodes. The query is evaluated by iterating over the nodes in postorder, skipping the Boolean nodes. In contrast to Algorithm 8 at Page 22, only those postings are fetched from the index that are needed for the particular join. The functions `compute_sum` and `compute_min`, which implement the Boolean operations on nodes, are integrated into the function `join_path` tightly. In this way, a number of iterations through the postings are avoided. The query processor provides a simple command line interface as well as an application programming interface that is used by web connector.

Recall that the result of the query-evaluation algorithm is a posting in which each entry represents the root of a query embedding. The query processor additionally computes the skeleton of the cheapest transformed query for each entry . A query skeleton is a tree of data nodes that are the images of an embedding of a transformed query. The query skeletons assigned to the entries of the resulting posting are used by the *abstract generator*. This module parses the matching XML document (which resides in the file system), and uses the node numbers of the query skeleton to find the element, attribute, or keyword belonging to the number. From the matching document part(s) an abstract is generated. An abstract consists of the highlighted matches together with a match context. The match context of a keyword are the k keywords preceding the match and the k following keywords, where k is a number specified by the database administrator. The match context of an attribute name is the attribute value the name of the element it belongs to. For every matching element name, some or all attributes are included in the match context. If a query leaf maps to an element name then a part of the text included in the element is added to the match context. All abstracts belonging to posting entries of a certain result window are created and retrieved to the user. During the first tests it emerged that the reparsing of the XML documents in order to generate abstracts is a performance bottleneck of the system. In the future, we plan the store the XML documents in a repository that provides fast access to a document subtree given a node number.

The web connector receives the user queries, passes them to the query processor, generates documents that contain the results of the search, and sends the documents back to the web client. Both HTML and XML is supported as format of the results. To speed up the connection between the web module and the HTTP daemon, we use an implementation of the FastCGI protocol [Bro96].

All server-side components of our system are implemented in C++ and have been tested successfully under Sun Solaris and Linux.

7.2. The Graphical Query Editor

Although the focus of this paper is on query *evaluation*, we claim that the usefulness of a retrieval system depends on the simplicity of query *formulation* as well. A non-expert user will neither be able to use the syntax of a formal query language nor will she know all element and attribute names occurring in a semistructured database. However, we also claim that a user interface that displays the entire database schema (which may be represented by one or more DTDs) will lead to an *information overload*. Therefore, the structure of the database should be presented in a simplified form.

Our prototype provides – besides of a simple HTML form field for specifying an approXQL query – a graphical query editor implemented as Java applet (see Figure 10). The basic idea behind this editor is to show a list of all element and attribute names that can be used in the query and to adapt the list to the current query context. In the simplest case, the list of names consists of all different element and attribute names found in the indexed documents. The database administrator can modify this list in order to ease query formulation: First, she can remove all names from the list which are not meaningful in typical user queries. For instance, the administrator may drop list and table constructors, elements that emphasize parts of the text, or attributes that contain information to control applications. Second, the administrator can replace element or attribute names by more descriptive labels. For instance, she can replace the label cd by compact disc. The

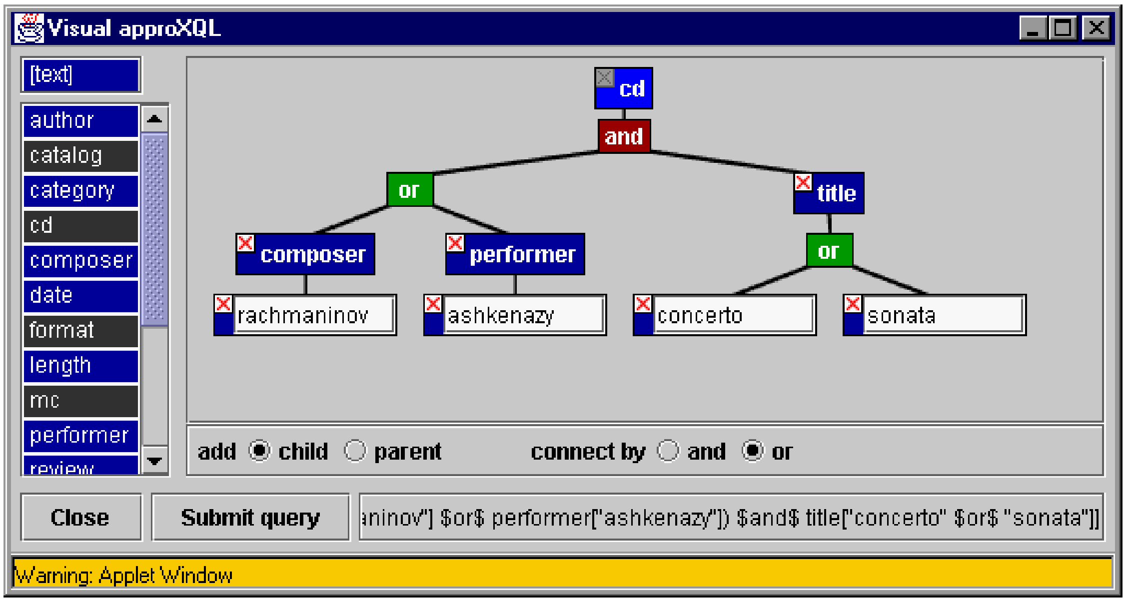


Figure 10: The graphical query editor.

query processor maps the synonyms to the original names using *zero-cost renamings*. To construct a query, the user selects a name from the list. The editor displays the selected name as labeled node in the tree panel. Depending on the construction mode ('add parent' or 'add child') the list of names is adapted so that all names not reachable from the selected name are disabled. The reachability information is provided by a strong DataGuide[GW97], which is requested from the approXDB server. Text fields can be added to the query whenever the current query node has a *text* descendant in the schema. The constructed query – which is additionally displayed in approXQL syntax – is then submitted to the approXDB server.

Currently, the graphical query editor supports only the basic syntax of approXQL introduced in Section 2. We plan to extend the editor so that the user can specify relaxations and enforcements of query transformations (see Section 5).

8. Related Work

Our work has three related areas: distance metrics for trees, query languages for XML, and structured queries in information retrieval.

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 the 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. All tree similarity measures we know do not take 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 the 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 are polynomial with respect the number of data nodes. We believe that algorithms that touch every data node are not usable for large databases.

Many query languages for semistructured data and XML in particular have been developed during the last years (see [FSW99, BC00] for surveys). All languages support operators that allow to skip certain parts of the data. However, they require that the user knows which parts have to be skipped.

To the best of our knowledge, XXL [TW00] and ELIXIR [CK01] are the only XML query languages stemming from the database community 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. Both languages do not allow partial structural matches.

There are also many proposals of query languages in the field of information retrieval. Here, the endeavor is to integrate content and structure in the query answering process (see [NB96, BR99] for surveys). Only few proposals exist that consider the similarity between queries and documents.

Navarro et. al. proposed to use result ranking for structured queries [NBVF98]. The authors introduced a simple query language and a generic ranking model. Queries are logical formulas consisting of presence and inclusion operators. The score computed for each document is the sum of all occurrences of the content terms specified by the query.

At the same time, a logic-based retrieval model for multimedia objects has been proposed [FGR98]. Aspects of this model are adopted by the XML query language XIRQL [FG00]. The language 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 [Röl99]. Like all other languages mentioned, XIRQL does not allow partial structural matches.

The main differences to the approach described in [SM00] are the scope (text centric versus data centric documents), the valuation model (rewarding versus penalizing), and the operators (weighted sum versus `and`/`or`).

9. Conclusion and Future Work

In this paper, we introduced the design and implementation of the pattern-matching language approXQL. An approXQL query retrieves exact matches but finds also results that are similar to the structure of the query. A subtree of an XML document is considered to be similar to the query if there is a sequence of basic query transformations such that the transformed query matches the subtree exactly. Each of the basic query transformations *insertion*, *deletion*, and *renaming* has a cost; the total cost of a sequence of transformations determines the degree of similarity between the original query and the result.

We think that cost-based query transformations are a powerful means to find approximate matches in data-centric XML documents. By adjusting the costs of the basic transformations, our approach can be easily adapted to different types of document content. However, the development of domain-specific rules for choosing basic transformation costs is a topic of future research. Our endeavor is to find algorithms that derive the similarity between document parts directly from the data. Possibly, techniques developed in the field of data mining can be tailored to fit our needs.

The query-evaluation algorithm presented in this paper must compute all results in order to retrieve the best n. Despite the good time complexity of the algorithm, the answering time may be bad for large databases. To improve the efficiency, we plan to estimate the best query transformations using data regularities captured by a (slightly modified) DataGuide [GW97].

An important topic of future research is the enhancement of the query language. As a replacement for the `text()` operator we intend to use a `data()` operator, which matches text as well as numeric data using *type coercion* [AQM+97]. The new operator should not only support equality tests but also less-than and greater-than comparisons. In addition, a future version of the approXQL should be able to cope with XML documents that have links. In particular, we plan to track links that model *containment*.

In the long term, we plan to combine our cost-based model with frequency-based models like the approach presented in [SM00]. The cost-based approach is well suited for data-centric documents where the distribution of terms is insignificant. For text-centric documents, on the other hand, the distribution of terms in document parts is certainly of importance. It is still an open problem whether both models can be integrated tightly. However, the cost-based model as well as the frequency-based approach can be implemented using the same framework, which essentially consists of operations on postings. We plan to realize both models in the same query processor, so that the interpretation of approXQL queries can be easily adapted to the type of document collection searched.

References

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

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

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

[Bro96] M. R. Brown. FastCGI: A high-performance web server interface, April 1996. http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm.

[CK01] T.T. Chinenyanga and N. Kushmerick. Expressive and efficient ranked queries for XML data. In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB'01)*, Santa Barbara, USA, May 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.

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

[FGR98] N. Fuhr, N. Gövert, and T. Rölleke. DOLORES: A system for logic-based retrieval of multimedia objects. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 257–265, Melbourne, August 1998.

[FSW99] M. Fernandez, J. Siméon, and P. Wadler. XML and query languages: Experiences and examples. http://www-db.research.bell-labs.com/user/simeon/xquery.ps, September 1999.

[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.

[Meg98] D. Megginson. SAX 1.0: The simple API for XML, May 1998. http://www.megginson.com/SAX/index.html.

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

[NB96] G. Navarro and R. Baeza-Yates. Integrating content and structure in text retrieval. *SIGMOD Record*, 25(1):67–79, March 1996.

[NBVF98] G. Navarro, R. Baeza-Yates, J. Vegas, and P. de la Fuente. A model and a visual query language for structured text. In *5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, Sta. Cruz de la Sierra, Bolivia, September 1998.

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

[Röl99] 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.

[SM00] T. Schlieder and H. Meuss. Result ranking for structured queries against XML documents. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, Zurich, Switzerland, December 2000.

[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.

[XER00] Xerces XML parser for C++. The Apache Software Foundation, 2000. http://xml.apache.org/xerces-c/.

A. Syntax of ApproXQL

```
       Query  ::=  LabelExpr Containment?
 Containment  ::=  '/' Expression | '[' Disjunction ']'
  Expression  ::=  Query | PhraseExpr
 Disjunction  ::=  Conjunction ( '$or$' Conjunction )*
 Conjunction  ::=  Group ( ConjOperator Group )*
ConjOperator  ::=  '$and$' | '$followedby$'
       Group  ::=  Expression | '(' Disjunction ')'
   LabelExpr  ::=  InsModifier?  LabelDef RenModifier?  DelModifier?
    LabelDef  ::=  LabelGroup | Label
  LabelGroup  ::=  '(' Label ( '|' Label )* ')'
       Label  ::=  WORD
  PhraseExpr  ::=  InsModifier?  PhraseSel RenModifier?  DelModifier?
   PhraseSel  ::=  ( SelFunction '=' )?  PhraseDef
   PhraseDef  ::=  PhraseGroup | Phrase
 PhraseGroup  ::=  '(' Phrase ( '|' Phrase )* ')'
      Phrase  ::=  '"' WORD ( WORD )* '"'
 SelFunction  ::=  'text()' | 'content()'
 InsModifier  ::=  '*' | '!'
 RenModifier  ::=  '*' | '!'
 DelModifier  ::=  ':'  ( DelCost | '*' | '!'  )
     DelCost  ::=  ( '+' | '-' ) NUMBER
```