Chapter 5

Querying by Approximate Tree Embedding

In the definition of the objectives for this thesis, we demanded that the interpretation of approXQL queries is vague. A query must retrieve not only exact matches, but also results with a structure and content similar to the selection conditions specified by the query. In this chapter, we present an innovative semantics for approXQL designed to meet these requirements. The semantics defines the set of valid results of a given query (represented by a set of query trees) and a given collection of XML documents (represented by a data tree containing logical documents). Its key concept is a new similarity measure that valuates structural deviations between queries and logical documents. The fewer deviations that exist, the more similar the query and the document are considered to be, and the higher is the position of the document within the list of results. With this interpretation a query selects all exactly matching documents — but it does not fail if it matches a document only partly, if its structure differs from that of a document, if the requested content exists in another structural context than specified, or if only content related to the specified one exists.

In the following section, we give an overview of our model to interpret approXQL queries. In Section 5.2, we provide definitions related to the exact interpretation of a query. Starting in Section 5.3, we present our approach towards finding results similar to a query. First, we define query transformations as a means to compensate the deviations between a query and a document. Second, in Section 5.4, we discuss several variants to assign costs to query transformations. Third, in Section 5.5, we show how query transformations can be composed in order to calculate similarity scores for the logical documents. In Section 5.6, we summarize the process of interpreting an approXQL query by means of an example. We conclude the chapter with a review of related work.

5.1 The Interpretation of approXQL Queries: An Overview

An approXQL query, given in its separated representation, is a set of query trees. Each tree represents a conjunct of the query. A query can be answered in an *exact* sense if at least one of the query trees can be embedded into the data tree. An embedding is a function that maps a query tree to trees included in the data tree. The function ensures a type-correct mapping of the query-tree nodes, fulfills the selection predicates, and preserves the parent-child relationships between the nodes. Each logical document (subtree of the data tree) in which a query tree can be embedded is considered to be a result of the query.

However, our goal is to find not only exact matches, but also results that are *similar* to the query. Our notion of similarity between a query and a logical document is defined as the degree of deviation between the structure of the query and the structure of the document. To measure this similarity, we apply sequences of basic transformations to the query trees in the separated representation of the query. We insert and delete nodes, permute pairs of nodes, and change the values of nodes. The basic transformations and the composition of transformations to sequences are restricted in order to meet two requirements: First, all transformations of a query tree must be intuitive and must result in query trees that describe a similar information need as the original one. Second, it must be possible to execute a query in polynomial (ideally sublinear) time with respect to the size of the query and the data tree.

We assign a cost to each basic transformation. The total cost of a sequence of basic transformations applied to a query tree is called the embedding cost. It determines the similarity between the original query and the logical documents in which the transformed query tree can be embedded. The costs of the basic transformations are the free parameters of our model. We assume that the costs are defined by a domain expert.

To answer an approXQL query, we create the set consisting of all trees that can be derived from the query trees in the separated representation via transformation sequences. This set is called the *closure* of the query. For each logical document of the data tree, we select the subset of query trees in the closure that can be embedded into the document. If more than one transformed query tree has embeddings, we choose the one with the lowest embedding cost. We use the embedding costs as similarity scores, rank the results by increasing costs, and output the best n results.

Figure 5.1 on the next page illustrates the interpretation of an approXQL query. Note that the figure does not show a practical query-evaluation method, because the closure is usually an infinite set. The efficient evaluation of queries is the topic of Chapters 6, 7, and 8.

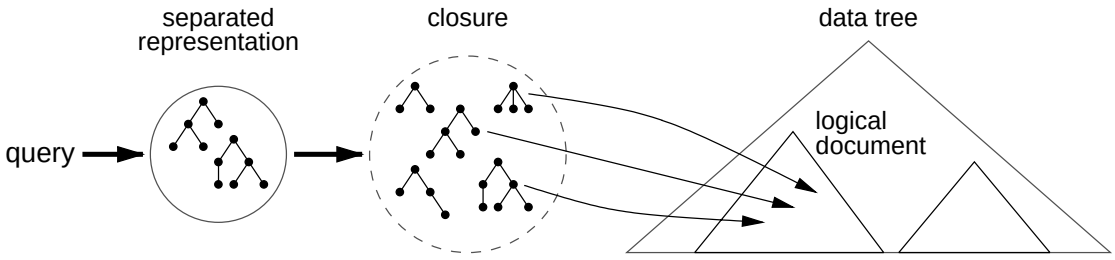


Figure 5.1: The interpretation of an approXQL query. First, the query is decomposed into its separated representation. Then, the closure of transformed query trees is derived. For each logical document, all embeddable query trees are selected. The tree with the lowest embedding cost determines the score of the document.

5.2 The Tree-Embedding Formalism

In Chapter 4, we have shown how to interpret a collection of XML documents as a single type-value tree, and how to model an approXQL query as a set of query trees. With this interpretation, we can map the problem of answering a query to the problem of embedding a query tree into a data tree:

Definition 5.1 (Embedding) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree and* $T_D = (N_D, E_D, r_D, type_D, value_D)$ *be a data tree. A mapping* $f : N_Q \to N_D$ *is called an embedding of* T_Q *into* T_D *if and only if for all* $u_Q, v_Q \in N_Q$ *holds:*

1. $u_Q = v_Q \Rightarrow f(u_Q) = f(v_Q)$ *(*f *is a function),*
2. $type_Q(u_Q) \preceq type_D(f(u_Q))$ *(*f *is type preserving),*
3. $pred_Q(u_Q)(value_Q(u_Q), value_D(f(u_Q)))$ *(*f *fulfills the selection predicates),*
4. $(u_Q, v_Q) \in E_Q \Leftrightarrow (f(u_Q), f(v_Q)) \in E_D$ *(*f *is parent-child preserving).*

The definition of the embedding function 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 siblings. We discard the injectivity property of the path inclusion problem in order to get a function that is efficiently computable.[1]

The second condition of Definition 5.1 postulates that the type of a query-tree node is either equal to or a supertype of the type of its image. In this way, the embedding function

[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 5.3.3, would lead to the *unordered tree inclusion problem* that is NP-complete [Kil92].

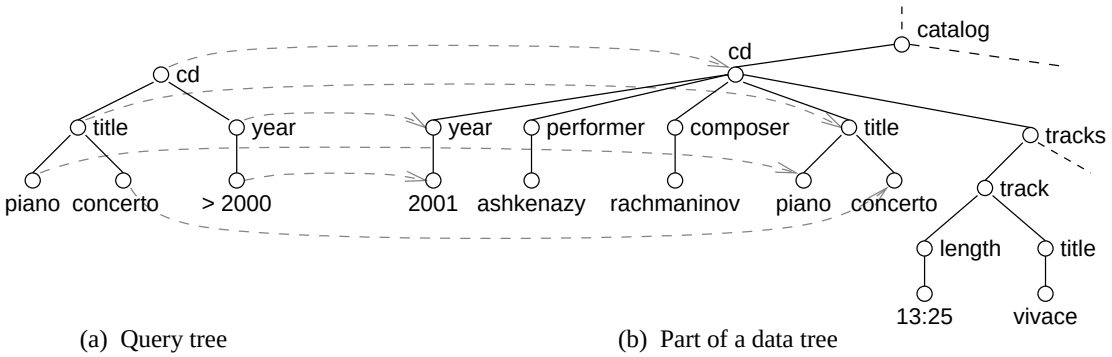


Figure 5.2: Embedding of a query tree into a data tree.

implements the typing rule of approXQL introduced in Section 4.2, which states that a more general type can be mapped to a more special type, but not vice versa. The third condition should be read as follows: The function $pred_Q$ of the query tree returns the selection predicate of node u_Q. This predicate is applied to the value of u_Q and to the value of the data-tree node $f(u_Q)$ to which u_Q is mapped. The condition is true if the selection predicate evaluates to true.

Figure 5.2 shows an embedding of a query tree (left) into a data tree. The dashed arrows symbolize the function, which preserves the types (not shown in the figure) and the parent-child relationships. All query-tree predicates are equality tests, except that of the lower-right node. The function $pred_Q$ of this node returns the binary predicate represented by the symbol ">". The predicate is applied to the corresponding integers and evaluates to true: $>(2000, 2001) = 2001 > 2000$.

Definition 5.2 (Match, Embedding image, Embedding root, Result) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree,* $T_D = (N_D, E_D, r_D, type_D, value_D)$ *be a data tree, and* f *be an embedding of* T_Q *into* T_D*. Each node* $f(u_Q)$*,* $u_Q \in N_Q$ *is a match;* $f(r_Q)$ *is the embedding root. The included tree* $T'_D = (N'_D, E'_D, r_D, type_D, value_D)$*, where*

$$\begin{aligned} N'_D &= \{\, f(u_Q) \mid u_Q \in N_Q \,\}, \\ E'_D &= \{\, (f(u_Q), f(v_Q)) \mid (u_Q, v_Q) \in E_Q \,\}, \end{aligned}$$

is the embedding image of f*. The logical document with the root* $f(r_Q)$ *is a result.*

In Figure 5.2, all nodes with incoming arrows are matches of query-tree nodes. The data-tree node with the value cd is the embedding root. The subtree rooted at the cd node is the result of the embedding. All matches, together with their connecting edges, form the embedding

image. Note that several results may exist for a fixed query tree and a fixed data tree, and several embeddings may lead to the same result.

5.3 Basic Transformations of Query Trees

The tree-embedding formalism allows exact embeddings only. To find results similar to the query, we use *basic transformations* of query trees. A basic transformation is a modification of a query tree by inserting a node, deleting a node, permuting nodes, or changing the value of a node. We write $T_Q \Rightarrow T'_Q$ to denote the basic transformation of a query tree T_Q to T'_Q. Each basic transformation has a cost. In Section 5.4, we discuss several variants for binding costs to transformations. For the moment, we assume that the costs are bound to the values of the nodes involved in basic transformations.

5.3.1 Deletions

The deletion of *inner* query-tree nodes is motivated by the observation that the hierarchical structure of XML data typically models containment relationships. The deeper an element resides in the data 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). Assume that users search for CD tracks with the title concerto. It is allowed to delete the title node in order to move to the more general context track in which the term concerto is searched, to delete the node track in order to search the term in titles of CDs, or to delete both title and track in order to search the term in the context of an entire CD.

The deletion of query-tree *leaves* is an adoption of "coordination level matching" [SM83], 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 scores, documents containing $n-1$ terms get the second-highest scores, and so on. This model can be elegantly combined with the model for deleting inner query-tree nodes:

Definition 5.3 (Deletion) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree, and* $v_Q \in N_Q$ *be a node such that* delres $\notin mod_Q(v_Q)$. *The deletion of* v_Q *is a transformation of* T_Q *to* $T'_Q = (N'_Q, E'_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *such that*

$$N'_Q = N_Q \setminus \{v_Q\},$$

$$\begin{aligned} E'_Q &= E_Q \setminus \{\, (u_Q, w_Q) \mid (u_Q, w_Q) \in E_Q \wedge (u_Q = v_Q \vee w_Q = v_Q) \,\} \\ &\quad \cup \{\, (u_Q, w_Q) \mid (u_Q, v_Q) \in E_Q \wedge (v_Q, w_Q) \in E_Q \,\}. \end{aligned}$$

The definition includes two restrictions: First, the deletion of a node is not permitted if it specifies a deletion restriction (delres). Second, it is not permitted to delete the root of a query tree, because this would create two or more separated query trees that asked for smaller result granularities than the original query tree did (e.g., for titles and years instead of CDs).

5.3.2 Permutations

The permutation of nodes in a query tree is motivated by the concept of *flexible queries* introduced by Kanza and Sagiv [KS01]. Queries are flexible in the sense that nodes on query paths may be permuted. There are two motivations for these permutations: First, the users may not know the containment relationships in a collection. For example, they expect documents with information about CDs, where each CD lists all its composers. They pose the query

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

However, the document collection actually stores information about composers, and for each composer, it lists all of the CDs with their works. For this collection, the query

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

is correct. The latter query can be derived from the former one by exchanging the selectors cd and composer, but keeping all other containment relationships.

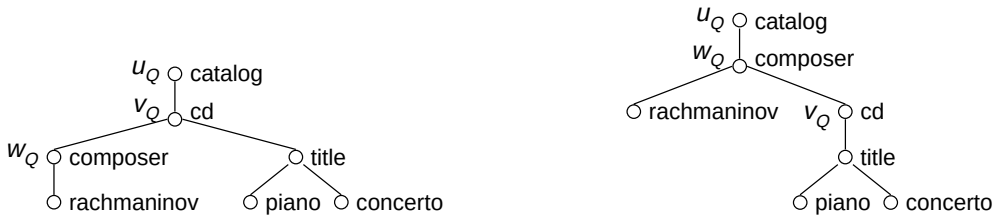
The second reason for permutations is the heterogeneity of a document collection. Within the same data tree, there may be information about CDs and their composers, and about composers and their CDs. Such heterogeneity emerges especially when documents belonging to different DTDs exist within a collection.

Definition 5.4 (Permutation) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree and* $(v_Q, w_Q) \in E_Q$ *be an edge such that* $struct \preceq type_Q(w_Q)$*. The permutation of* v_Q *and* w_Q *is a transformation of* T_Q *to* $T'_Q = (N_Q, E'_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *such that*

$$E'_Q = \begin{cases} (E_Q \setminus \{(u_Q, v_Q), (v_Q, w_Q)\}) \cup \{(u_Q, w_Q), (w_Q, v_Q)\} & \text{if } \exists (u_Q, v_Q) \in E_Q, \\ (E_Q \setminus \{(v_Q, w_Q)\}) \cup \{(w_Q, v_Q)\} & \text{else.} \end{cases}$$

The first rule captures the case where v_Q is not the root of the query tree. Here, the edge from the parent u_Q of v_Q must be removed, and replaced by a new edge from u_Q to the permuted node w_Q. The second rule captures the simpler case $v_Q = r_Q$.

Consider the query tree depicted in Figure 5.3(a), and assume that the permutation of the nodes v_Q (with value `cd`) and w_Q (with value `composer`) is defined. First, the edge (v_Q, w_Q) is replaced by the edge (w_Q, v_Q), and second, the edge (u_Q, v_Q) is replaced by the edge (u_Q, w_Q). No other edges are touched, so that all other hierarchical relationships of the query tree remain unchanged.



(a) Original tree

(b) Permuted tree

Figure 5.3: Permutation of two nodes (v_Q, w_Q) in a query tree.

The main differences from the model proposed by Kanza and Sagiv are that we assign costs to permutations, and that we do not allow arbitrary permutations. Only predefined node pairs may be permuted. For example, the domain expert may define that nodes of type *element* having the values cd and composer may be permuted (because the data tree contains both variants), but all other node pairs must remain unchanged. Other differences from the model of Kanza and Sagiv are that we do not require the query root to be mapped to the root of the data tree, and that we only allow the permutation of nodes that have the type *struct* or one of its subtypes.

5.3.3 Insertions

A node insertion creates a query that expects the matches of a query subtree in a more specific context. For example, the insertion of two nodes with the values tracks and track, respectively, between the query-tree nodes cd and title depicted in Figure 5.2(a) switches from the more general context *CD title* to the more specific context *CD track title*.

Definition 5.5 (Insertion) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree,* $v_Q \notin N_Q$ *be a node, and* $(u_Q, w_Q) \in E_Q$ *be an edge such that* insres $\notin mod_Q(w_Q)$. *An*

insertion of v_Q between u_Q and w_Q is a transformation of T_Q to $T'_Q = (N'_Q, E'_Q, r_Q, type'_Q, value'_Q, pred'_Q, mod'_Q)$ such that

$$\begin{aligned} N'_Q &= N_Q \cup \{v_Q\}, \\ E'_Q &= (E_Q \setminus \{(u_Q, w_Q)\}) \cup \{(u_Q, v_Q)\} \cup \{(v_Q, w_Q)\}. \end{aligned}$$

The functions $type_Q$, $value_Q$, and $pred_Q$ are extended by mappings of v_Q to its type, to its value, and to its predicate, respectively. The result of $mod'_Q(v_Q)$ is an empty set.

The definition includes four restrictions: First, the insertion of a node is not permitted if the child node of the edge to be replaced specifies an insertion restriction (insres). Second, it is not permitted to append new leaves, because this would create a query that asked for information not requested by the users. Third, it is not permitted to append a new root, because this would change the granularity of the results (e.g., from CDs to CD collections). Fourth, each inserted node replaces exactly one edge. Although this is a formal restriction of the general case that allows the replacement of multiple edges, it does not significantly change the semantics of the general case: Whenever a single node replaces k edges in the general model, we insert k new nodes with the same type and value. Because embeddings are not required to be injective, the k new nodes can be mapped to the same data-tree node. This restriction is one of the reasons (non-injective embeddings and ordered transformation sequences are the others) that the evaluation of a query in our model has polynomial time complexity instead of being NP-hard [Kil92] (if we considered insertions only) or even MAX SNP-hard [ZSS92] (if deletions and value changes were permitted in addition).

A sequence of cost-based node insertions can be considered as a counterpart to the "//" operator of XPath. There are, however, two major differences: First, the users must explicitly apply the XPath operator "//" instead of "/" in order to skip an arbitrary number of data-tree nodes. To use this operator, the users must guess that some nodes have to 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 users do not need to know that some nodes have to be skipped. Second, in our approach the insertion of a node imposes a cost. This cost penalizes for the transition to a more specific context.

5.3.4 Value Changes

Changing the value of a query-tree node shifts the search space of the query subtree rooted at this node. For example, changing the value of the query root from cd to mc obviously shifts

the search space from CDs to MCs; the change of title to category shifts the context in which the keywords piano and concerto are expected. Similarly, the change of the keyword concerto to sonata shifts the search space of the text selector. If the selection predicate does not test for equality, then the old and new search spaces may overlap. For example, if a query selects CDs appeared after 2000, then the change of the value from 2000 to 1999 *broadens* the search space.

Definition 5.6 (Value change) *Let* $T_Q = (N_Q, E_Q, r_Q, type_Q, value_Q, pred_Q, mod_Q)$ *be a query tree, and* $u_Q \in N_Q$ *be a node such that* valres $\notin mod_Q(u_Q)$*. The change of the value of* u_Q *from* α *to* α' *(*α *and* α' *must have the same domain) is a transformation of* T_Q *to* $T'_Q = (N_Q, E_Q, r_Q, type_Q, value'_Q, pred_Q, mod_Q)$ *such that*

$$\forall v_Q \in N_Q : value'_Q(v_Q) = \begin{cases} \alpha' & \text{if} \quad v_Q = u_Q, \\ value_Q(v_Q) & \text{else.} \end{cases}$$

All value changes must be predefined for a given query and a given data tree. For instance, an allowed value change of the value title to category may be indicated by a tuple (title, category, 4), where 4 is the cost of the basic transformation. If we allowed default changes of values, then we would have $O(|N_D|)$ possible matches for each query-tree node, where $|N_D|$ is the number of nodes in the data tree.

Value changes are a very powerful concept. Pairs of alternative values may be specified by a domain expert with respect to a *particular* collection of XML documents. For example, the domain expert may figure out that title and category elements typically have related content, and may reflect this similarity in content by a cost-based value change. Alternatively (or in addition to), the similarity between values may be derived from a *thesaurus* or an *ontology*. Value changes — in particular zero-cost value changes — are very helpful for the fast integration of document collections in which the elements and attributes carry different names, but contain similar content. For example, the author element of an article may have the names author or article_author; its name may belong to another language than English; or it may have a "meaningless" shorthand name like AU.

5.4 Assigning Costs to Basic Transformations

Each basic transformation modifies a given query tree; the cost of the applied basic transformation valuates the similarity between the original and the modified query tree. The more

similar the embeddings of both query trees are, the lower is the cost of the transformation. In the most general case, the cost of a basic transformation is a function that maps pairs of query trees to numbers:

Definition 5.7 (Transformation costs) *Let $\mathbb{T}$ and $\mathbb{T}'$ be sets of query trees such that for each $(T_Q, T'_Q) \in (\mathbb{T} \times \mathbb{T}')$ holds $T_Q \Rightarrow T'_Q$. The transformation costs are defined by a function*

$$cost : \mathbb{T} \times \mathbb{T}' \rightarrow \mathbb{R}^+,$$

where $\mathbb{R}^+$ denotes the non-negative real numbers.

We use the notation $cost(T_Q \Rightarrow T'_Q)$ to refer to the cost of the basic transformation $T_Q \Rightarrow T'_Q$. The general cost-assignment function cannot be used in practice, because we would have to define costs for all query trees that may appear, and for all possible transformations of them. Therefore, we discuss more restricted variants of assigning costs. The costs of basic transformations should depend on the properties of the involved nodes, which include (i) the types and values of the nodes, (ii) the positions of the nodes in the query tree, and (iii) the properties of the matches of the nodes.

Type-value–specific costs. This method binds the costs to the types and values of the nodes. If two nodes have the same value but a different type, then the cost may differ. In our example depicted in Figure 5.2 on page 54, CD titles and CD track titles would have the same insertion and deletion costs. The change of the value title to category would impose the same cost for CD titles and CD track titles.

Role-specific costs. This variant is a specialization of type-value–specific costs. It includes the different roles of a node with a given type and value. For example, a node of type *element* and value title may get different costs depending on its role as CD title, MC title, or CD track title.

DTD-specific costs. This is also a specialization of type-value–specific costs. In a DTD, each element name may appear only once, and each element may only have one content model. This content model describes the semantics of the element — independently from the place the element appears in an XML document that instantiates the DTD. The DTD-specific cost assignment adds the uniform resource identifier (URI) of the DTD as a prefix to each element name, and the URI and the element name as prefix to each attribute name. In the data tree constructed for documents of different DTDs, an attribute name title will appear, e.g., as DTD1:cd:title and as DTD2:cd:title. To

map a query node with value title to a data-tree node, its value must be changed to DTD1:cd:title or DTD2:cd:title, respectively. Both value-changes may get different costs. The same model can be applied to names that appear within a namespace [BHL99].

The role-specific cost assignment and the DTD-specific cost assignment can be used in combination. For the rest of the thesis, we assume *type-value–specific costs* as the simplest method of assigning costs.

5.5 Approximate Query Answering

In this section, we formally define our notion of approximate query answering. We first define the notion of a transformation sequence:

Definition 5.8 (Transformation sequence) *Let T_Q^0 be a query tree. A transformation sequence $T_Q^0 \Rightarrow T_Q^1 \Rightarrow \ldots \Rightarrow T_Q^n$ is a series of basic transformations, where all deletions precede all permutations, all permutations precede all value changes, and all value changes precede all insertions.*

The order of transformation sequences has an intuitive semantics: All destructive operations (deletions) precede all global restructurings (permutations), all global restructurings precede all local restructurings (value changes), and all local restructurings precede all constructive operations (insertions). This order forbids many sequences that were possible in an unrestricted model. However, most of those forbidden combinations would be redundant or unintuitive: It would be redundant to delete a previously inserted node or to delete a node whose value has previously been changed. It would also be redundant to change the value of a previously inserted node — we would yield the same result if we inserted a node with the appropriate value. Furthermore, it would be unintuitive to delete nodes that participated in a previous permutation. Finally, permutations and value changes are commutative, so they can be applied in arbitrary order. The order of transformation sequences is no substantial restriction on our model, and it is one of the key concepts that allows an implementation with a favorable time complexity (see Chapter 6).

Definition 5.9 (Embedding cost) *Let T_Q^0 be a query tree, and $T_Q^0 \Rightarrow T_Q^1 \Rightarrow \ldots \Rightarrow T_Q^n$ be a transformation sequence. The embedding cost of T_Q^n is defined as*

$$embcost(T_Q^n) = \sum_{i=1}^{n} cost(T_Q^{i-1} \Rightarrow T_Q^i).$$

To establish the relationship between an approXQL query and the set of logical documents selected as results, we use the theoretical concept of a query *closure*. The closure of a query is the (infinite) set of query trees that can be derived from the trees in the separated query representation via transformation sequences.

Definition 5.10 (Query closure) *Let $\mathbb{Q}$ be the separated representation of a query Q. The closure of Q is defined as*

$$\mathbb{Q}^* = \mathbb{Q} \cup \{\, T'_Q \mid \exists T_Q \in \mathbb{Q} \text{ such that there is a transformation sequence } T_Q \Rightarrow \ldots \Rightarrow T'_Q \,\}.$$

To find all results of Q, we try to embed each query tree $T_Q \in \mathbb{Q}^*$ into the data tree T_D according to Definition 5.1 on page 53. Several transformed query trees may have embeddings with the same root, which means that they are embedded into the same logical document. For each logical document $T_D[u_D]$, where u_D is a particular data-tree node that defines the root of the document, we select the *best* query tree. The best query tree is the one that has the lowest embedding cost of all query trees in $\mathbb{Q}^*$ that have embeddings in $T_D[u_D]$. We say that this tree defines the *distance* between the query and the logical document $T_D[u_D]$:

Definition 5.11 (Approximate query-matching distance) *Let Q be a query, T_D be a data tree, and $T_D[u_D]$ be a logical document. Let $\mathbb{Q}^*$ be the closure of Q and $\bar{\mathbb{Q}}^* \subseteq \mathbb{Q}^*$ be all query trees that have embeddings into T_D such that $T_D[u_D]$ is the result. The approximate query-matching distance between Q and $T_D[u_D]$ is defined as*

$$dist(Q, T_D[u_D]) = \begin{cases} \infty & \text{if } \bar{\mathbb{Q}}^* \text{ is empty,} \\ \min\{\, embcost(T_Q) \mid T_Q \in \bar{\mathbb{Q}}^* \,\} & \text{else.} \end{cases}$$

Using the distance between a query and a logical document, we define the *all-results problem*:

Definition 5.12 (All-results problem) *Given a query Q and a data tree T_D, construct the set of all pairs consisting of the root of a logical document and an embedding cost:*

$$S = \{\, (u_D, c) \mid T_D[u_D] \text{ is a logical document } \wedge\, c = dist(Q, T_D[u_D]) \wedge c < \infty \,\}.$$

A pair $(u_D, c) \in S$ is called *node-cost pair*. Note that each pair in S contains the root of a logical document. Using the root, the document can be easily retrieved from the data tree. Because users are typically interested in the *best* results only, we define the *best-n-results problem*:

Definition 5.13 (Best-n-results problem) *Create a cost-sorted set of the n node-cost pairs in S that have the lowest embedding costs among all node-cost pairs in S.*

5.6 Example

To illustrate the interpretation of an approXQL query, we consider a simple conjunctive query, and allow only a very restricted set of basic transformations. We use the query

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

and the following basic transformations and costs:

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

All other basic transformations receive an infinite cost. In particular, it is not allowed to insert or permute nodes. As an additional simplification, we use the approXQL syntax to depict query trees in textual notation.

At first, we create the separated representation of Q. Because Q has no "`or`" operators, Q is already in hierarchical disjunctive normal form. Therefore, the separated representation of Q consists of a single query tree:

$$\mathbb{Q} = \{\, T_{Q_1} \} = \{\, \texttt{cd[title["piano" and "sonata"] and performer["rachmaninov"]]} \,\}.$$

In the second step, we derive the closure of Q from $\mathbb{Q}$:

$$\begin{aligned}\mathbb{Q}^* &= \{\, T_{Q_1}, T_{Q_2}, T_{Q_3}, T_{Q_4}, T_{Q_5}, T_{Q_6} \} \\ &= \{\, \texttt{cd[title["piano" and "sonata"] and performer["rachmaninov"]],} \\ &\quad\ \texttt{cd[title["piano"] and performer["rachmaninov"]],} \\ &\quad\ \texttt{cd[title["piano" and "concerto"] and performer["rachmaninov"]],} \\ &\quad\ \texttt{cd[title["piano" and "sonata"] and composer["rachmaninov"]],} \\ &\quad\ \texttt{cd[title["piano"] and composer["rachmaninov"]],} \\ &\quad\ \texttt{cd[title["piano" and "concerto"] and composer["rachmaninov"]]} \,\}.\end{aligned}$$

In the third step, we embed the query trees in $\mathbb{Q}^*$ into the data tree shown in Figure 5.2(b) on page 54. There is only one logical document in the depicted part of the tree — the subtree rooted at the node cd. The last two transformed queries in $\mathbb{Q}^*$ can be embedded into the document. According to our cost assignment, the query tree

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

has the embedding cost 13, because the keyword sonata has been deleted (cost 8), and performer has been changed to composer (cost 5). The last query in $\mathbb{Q}^*$,

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

has the embedding cost 8, because sonata has been changed to concerto (cost 3), and performer has been changed to composer (cost 5). It follows that the distance between the query and the logical document is $\min(13, 8) = 8$.

In step four, we choose the n subtrees (logical documents) with the smallest distances. In our example, there is only one document. Finally, we output the pair consisting of the document root and the distance.

5.7 Related Work

Recall from Section 2.8 that very few proposals for query languages and retrieval models are designed for similarity search in XML data with complex, heterogeneous structure. In most cases, the interpretation of queries is strict. Some approaches support flexible mappings, but do not valuate the similarity between queries and results (e.g., [Kil92, HWC+99, KS01]). Languages like XIRQL [FG01], ELIXIR [CK02], and XXL [TW02] valuate the similarity between keywords and names of elements and attributes. However, they do not support partial matches, automatic edge relaxation, or permutations.

The model presented by Amer-Yahia et. al. [ACS02] includes some of the properties of the approXQL semantics, but is restricted to conjunctive queries and to the structural parts of queries. It lacks permutations and supports only a single value-change per node (called generalization). Moreover, the valuation function for node insertions (called edge relaxations) is less general than ours: The total cost for node insertions depends only on the number of inserted nodes rather than on the properties of those nodes (reflected by insertion costs in our model).

In Section 2.8, we also pointed out that the classical tree-distance measures are of limited use for searching in XML data. We summarize the weaknesses: First, the measures are not designed for queries with conjunctive and disjunctive parts. Second, the only supported selection condition is the test for the equality of labels. Third, many measures are defined

for ordered trees. However, it cannot be expected that users will be aware of the order of elements, attributes, and even words in the data. Fourth, deletions and insertions are unrestricted, and are often not intuitive for the context of approximate XML queries. Fifth, renamings rely on the neighborhood of nodes rather than on the semantic closeness between the node labels. Sixth, the computation of the distance between unordered trees is an NP-hard problem. The proposed algorithms for computing the distance between ordered trees visit each node in both trees several times. The semantics of approXQL overcomes the weaknesses of tree-distance measures: First, queries may consist of both conjunctive and disjunctive parts. Second, several built-in selection predicates are supported; user-defined predicates can be added. Third, embeddings are unordered. Fourth, the query transformations are restricted in order to forbid unintuitive operations like deleting the query root or appending new leaves. Fifth, value changes rely on semantic closeness between words. Sixth, a query can be evaluated in polynomial time with respect to the size of the query. The only data-tree nodes that have to be visited are the matches of query-tree nodes (see Chapters 6–8).