# Chapter 7

# Schema-Driven Query Evaluation

An initial weakness of the direct query-evaluation method presented in the previous chapter is that we must compute *all* results in order to retrieve the *best $n$* ones, after sorting and pruning the result set. We simply do not know the cost of a logical document until we have applied the last plan operator, and because we cannot estimate the final cost in an early evaluation step, we do not know which documents can be discarded. A method to overcome the lack of knowledge about the final cost of a partially evaluated document is to involve additional information about common structures of the data tree. During the query evaluation, this additional information can help to determine to which extent the cost of the current document is expected to grow. In this chapter, we present a consequent realization of this idea: We use the path tree (schema) of the data tree to find the best $k$ query embeddings in cost-sorted order. The images of these embeddings are used as "second-level queries", which are executed against the data tree in order to find the best $n$ results. To find the best $k$ second-level queries, we make use of the algorithmic framework presented in the previous chapter. We apply few changes to the five set operators, but do not modify the algorithms for the construction and evaluation of query-execution plans.

The chapter is organized as follows: In the first section, we formally define path trees and investigate their relationships to data trees. Starting in Section 7.2, we present a method for evaluating a query with the help of a path tree. We first extend node-cost tuples so that they can not only track an embedding cost, but also the image of the embedding for which the cost has been calculated. Next, we show how the operators must be adapted to find the best $k$ second-level queries. In Section 7.3, we introduce an algorithm that finds all results of a given second-level query. The algorithms for the creation and evaluation of second-level queries are then integrated into an incremental query-evaluation algorithm that retrieves the best $n$

results of a query given an initial guess for $k$. This algorithm is presented in Section 7.4. In Section 7.5, we show how the incremental algorithm can be optimized by merging second-level queries with similar substructures. At the end of the chapter, we review related work.

## 7.1 The Relationship between a Data Tree and its Path Tree

In a data tree constructed for a collection of XML documents, many subtrees have similar structures. A collection of sound storage media may contain several CDs that all have a title, a composer, or both. Such data regularities can be captured by a *path tree*, which is similar to a DataGuide [GW97]. In this section, we investigate how the path tree of a data tree can be used to find all logical documents matching a query.

### 7.1.1 Path Trees and Node Classes

A path tree can be considered as a data tree in which sibling nodes with equal types and values are merged. To define a path tree formally, we need the notion of a *type-value path*:

**Definition 7.1 (Type-value path)** *A type-value path* $(\tau_1, \alpha_1).(\tau_2, \alpha_2)\ldots(\tau_n, \alpha_n)$ *in a type-value tree* $T = (N, E, r, \mathit{type}, \mathit{value})$ *is a sequence of type-value pairs, where each pair* $(\tau_i, \alpha_i)$ *represents a node* $u_i \in N$ *such that*

1. $u_1 = r$,
2. $\tau_i = \mathit{type}(u_i)$ *for all* $1 \leq i \leq n$,
3. $\alpha_i = \mathit{value}(u_i)$ *for all* $1 \leq i \leq n$, *and*
4. $(u_i, u_{i+1}) \in E$ *for all* $1 \leq i < n$.

Note that a type-value path starts at the root of a tree, but does not necessarily end at a leaf.

**Definition 7.2 (Path tree)** *The path tree* $T_P$ *of a data tree* $T_D$ *is a type-value tree such that (i) for each type-value path in* $T_D$ *there is exactly one equal type-value path in* $T_P$, *and (ii) for each type-value path in* $T_P$ *there is an equal type-value path in* $T_P$.

The path tree of a data tree is unique because the left-to-right order of sibling nodes is irrelevant. Figure 7.2 on the next page shows the path tree constructed for the data tree

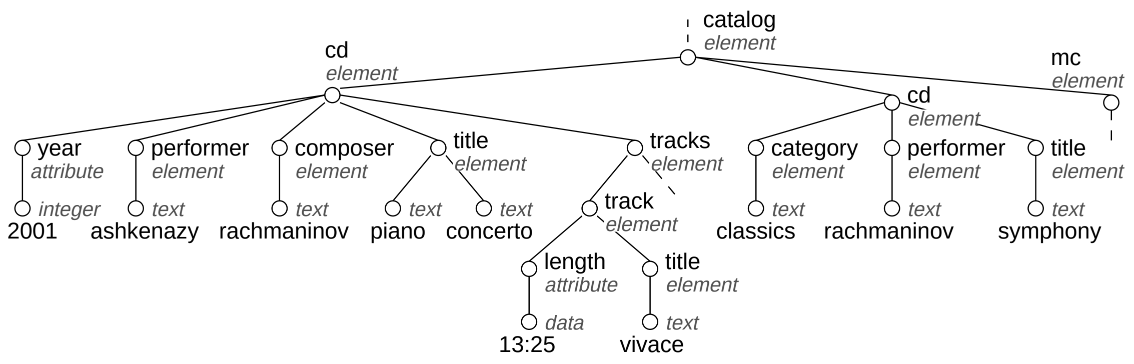

Figure 7.1: A part of a data tree.

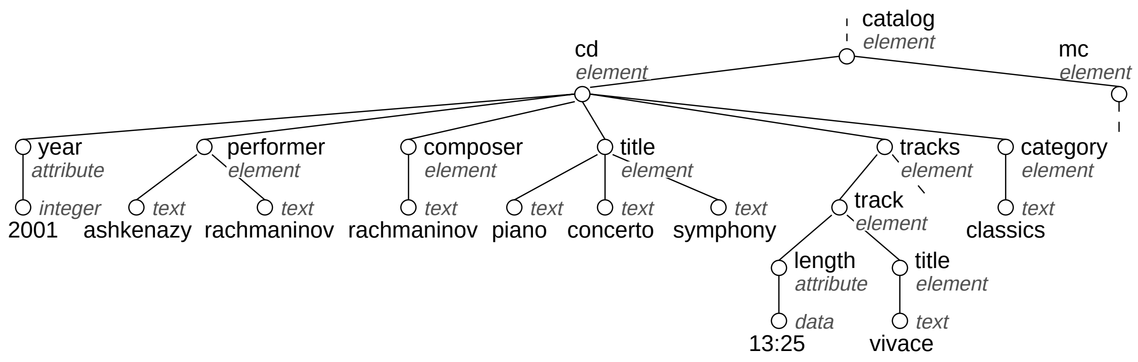

Figure 7.2: The path tree of the data tree depicted in Figure 7.1.

depicted in Figure 7.1. The number of nodes in this path tree is not significantly smaller than the number of nodes in the corresponding data tree, because the data tree does not have repeated structures, and because all distinct leaf values are represented in the path tree. The explicit representation of leaf values is a simplification that helps to explain our approach. In the next chapter, we introduce *compacted path trees*, which have identical properties with respect to the approximate query-matching distance, but have much smaller sizes.

**Definition 7.3 (Node class, node instance)** *Let $T_D$ be a data tree, and $T_P$ be its path tree. A node $u_P$ in $T_P$ is the class of a node $u_D$ in $T_D$ if and only if $u_P$ and $u_D$ are reachable by equal type-value paths. If $u_P$ is the node class of $u_D$, then $u_D$ is an instance of $u_P$.*

We use the notation $u_P = [u_D]$ to denote the node class $u_P$ of $u_D$, and the notation $instances(u_P)$ to refer to the set of instances of $u_P$. Each data-tree node $u_D$ has exactly one node class because there is only one type-value path from the root of the data tree to $u_D$.

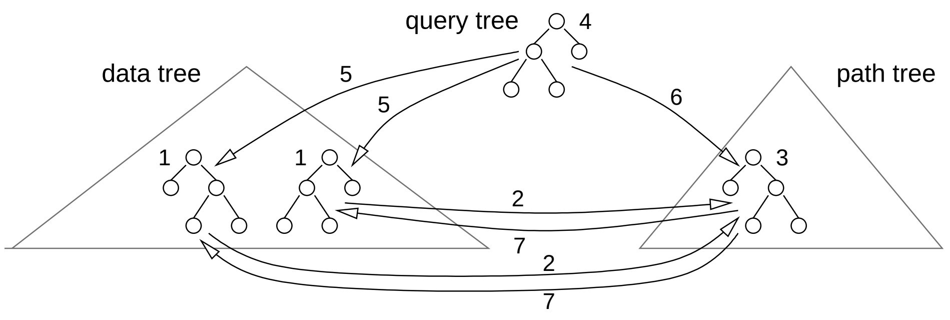

Figure 7.3: The relationship between a data tree and its path tree.

### 7.1.2 Using the Path Tree to find the Images of Data-Tree Embeddings

We prove that there are two equivalent ways of finding the images of data-tree embeddings: The direct embedding of query trees, and a two-level method that uses the path tree of the data tree in an intermediate step. Figure 7.3 gives an overview of our proofs. First, we show that each tree included in the data tree (1) has an embedding (2) into the path tree. The image (3) of the embedding is called a *tree class*. Several trees included in the data tree may have the same tree class. Next, we show that embeddings are transitive. Given a query tree (4) and embeddings in the data tree (5), we can also construct an embedding of the query tree in the path tree (6) so that the embedding image (3) is a tree class of (1). In the next step, we show that each tree class has reverse embeddings (7) that help to locate the instances of the tree class, and that only tree classes have reverse embeddings. It follows that we can find the images of the embeddings of a query tree $T_Q$ into a data tree by selecting the images of the path-tree embeddings of $T_Q$ and using them to find the images of the data-tree embedding of $T_Q$.

**Definition 7.4 (Tree class, tree instance)** *Let $T'_D$ be a tree included in $T_D$, and $T'_P$ be the image of an embedding of $T'_D$ into $T_P$. Then $T'_P$ is a tree class of $T'_D$ and $T'_D$ is a tree instance of $T'_P$.*

Each tree included in the data tree has a tree class. Intuitively, the construction of an embedding of an arbitrary tree included in the data tree into the path tree is possible because the path tree preserves all vertical relationships of the data tree as well as all horizontal relationships required by embeddings. Of course, not all horizontal relationships of the data tree are preserved; in particular, the order of siblings is lost and siblings with identical types and values are mapped to a single node in the path tree. But these are exactly the relationships not required by embeddings — by definition, an embedding is unordered and not injective.

**Lemma 7.1 (Tree-class existence)** *Let $T_D$ be a data tree, and $T_P$ be its path tree. Each tree $T'_D$ included in $T_D$ has a tree class in $T_P$.*

**Proof:** We prove the lemma by constructing an embedding $f$ of $T'_D$ into $T_P$ (see Definition 5.1 on page 53). We set $f(u_D) = [u_D]$ for each $u_D$ in the node set of $T'_D$. The mapping $f(u_D)$ is well-defined and is a type and value-preserving function (Definition 7.3). Therefore, $f$ fulfills the first three properties of Definition 5.1. Let $v_D$ be a child of $u_D$. Definition 7.2 states that each parent-child relationship in the data tree corresponds to a parent-child relationship in the path tree. That is, $[v_D]$ is a child of $[u_D]$, and therefore $f(v_D)$ is a child of $f(u_D)$. Thus, Property 4 of Definition 5.1 holds and $f$ is an embedding. $\square$

We cannot ensure the opposite direction: Not every tree included in the path tree is a tree class because the path tree contains sibling relationships that have no counterparts in the data tree. Consider the path tree depicted in Figure 7.2 on page 111. The included tree consisting of the cd node and the children composer and category is not a tree class because there is no CD in our catalog that has both a composer and a category. Moreover, a tree included in a data tree may belong to several tree classes. For example, a tree with the root value title and the leaf value piano may appear several times in both a data tree and its path tree. All tree classes of a tree included in a data tree share the same instances, and therefore only one of the tree classes has to be considered.

So far, we have shown that each tree included in the data tree has a tree class (although this tree class is not necessarily unique). An implication of this proposition is that each image of an embedding of a query tree $T_Q$ in the data tree has a tree class. We now need a way to find one of the tree classes of the embedding images of $T_Q$ without accessing the data tree. Once we have this class, we can use it to find the embeddings of $T_Q$ in the data tree. The following lemma helps to prove that we can in fact find a tree class using only the path tree.

**Lemma 7.2 (Embedding transitivity)** *Let $f$ be an embedding of a tree $T_A$ into a tree $T_B$, where $T'_B$ is the image of $f$. Let $f'$ be an embedding of $T'_B$ into a tree $T_C$, where $T'_C$ is the image of $f'$. Then exists an embedding $f''$ of $T_A$ into $T_C$ such that $T'_C$ is the image of $f''$.*

**Proof:** All four properties of Definition 5.1 are transitive. We can therefore construct an embedding $f''$ by defining $f''(u_A) = f'(f(u_A))$ for each $u_A$ in the node set of $T_A$. $\square$

Lemma 7.2 allows us to find the tree classes belonging to the images of all data-tree embeddings of a query tree $T_Q$: Whenever $T_Q$ has an embedding in the data tree, the image of the embedding has a tree class $T'_P$. Because embeddings are transitive, there is also an embedding of $T_Q$ into the path tree such that $T'_P$ is the image. Thus, if we have the images of all path-tree embeddings of $T_Q$, then we also have all tree classes belonging to the data-tree embeddings of $T_Q$. However, we are not interested in the tree classes of the images of data-tree embeddings, but rather in the images itself. The following lemma shows that all tree classes (any only tree classes) have reverse embeddings that help to locate their instances.

**Lemma 7.3 (Reverse embeddings)** *Let $T_D$ be a data tree, $T_P$ be its path tree, and $T'_P$ be a tree included in $T_P$. If $T'_P$ is a tree class of a tree $T'_D$ included in $T_D$, then there exists an embedding of $T'_P$ into $T_D$ such that the root of $T'_D$ is the root of the embedding image. If $T'_P$ is not a tree class of a tree included in $T_D$, then no embedding into $T_D$ exists.*

**Proof:** *Case one*: $T'_P$ is a tree class of $T'_D$. Let $f$ be the embedding that maps $T'_D$ to $T'_P$. We construct a reverse embedding $f'$ as follows: Each node $u_P$ in $T'_P$ is the image of a node $u_D$ in $T'_D$. If $u_P = f(u_D)$ is the image of only one node $u_D$, then we define $f'(u_P) = u_D$. Otherwise, if $u_P$ is the image of several sibling nodes in $T'_D$, we choose an arbitrary one (say $w_D$) and define $f'(u_P) = w_D$. Each node in $T'_P$ is mapped to a single node in $T'_D$. Therefore, $f'$ is a function. The properties 2, 3, and 4 of Definition 5.1 are equivalences so that for each pair $u_P, v_P$ holds $\mathit{value}(u_P) = \mathit{value}(f'(u_P))$, $\mathit{type}(u_P) = \mathit{type}(f'(u_P))$, and $(u_P, v_P) \in E'_P \Leftrightarrow (f'(u_P), f'(v_P)) \in E'_D$. It follows that $f'$ is an embedding of $T'_P$ into $T'_D$.
*Case two*: $T'_P$ is not a class of any tree included in $T_D$. Assume there was an embedding of $T'_P$ into $T_D$ with the image $T'_D$. Then we could construct an embedding $f$ that maps $T'_D$ to $T'_P$ in analogy to the construction of $f'$ described above (the construction is even simpler because no equal-valued siblings exist in the path tree). However, if $f$ existed then $T'_P$ would be a tree class. □

The lemmata imply a simple two-level evaluation method for a query tree: If we have a "primary" algorithm that finds the images of all of the embeddings of a query tree into a data tree, then we can use the same algorithm to find the tree classes of those images (Lemmata 7.1 and 7.2). However, the algorithm may also find embedding images in the path tree that are not tree classes. We additionally need a "secondary" algorithm that uses the image of a path-tree embedding as a "second-level query", and finds all reverse embeddings for that image. Because only tree classes have reverse embeddings, and for a given class a reverse embedding to each of its instances exist (Lemma 7.3), the two-level approach finds the images of all embeddings of the query tree into the data tree, and it finds only those.

In the following sections, we use the equivalence between direct and two-level embedding to implement efficient algorithms for the schema-driven evaluation of approXQL queries.

## 7.2 Finding Second-Level Queries in a Path Tree

In Chapter 6, we presented an algorithmic framework for the evaluation of an approXQL query with respect to a data tree. In this section, we extend this framework in a way that during the evaluation of a query not only the roots of the embeddings are found, but also the images of them. The extended framework is then used to find the images of path-tree embeddings (second-level queries). We do neither modify the construction of query-execution plans, nor the plan-evaluation algorithm. Instead, we apply changes to the basic level of the framework: We extend node-cost tuples so that they can store not only an embedding cost, but also the image of the embedding for which the cost has been calculated. However, the objective of this chapter is to retrieve only the *best* $n$ results. To find them, we need the $k$ second-level queries with the lowest embedding cost. (In Section 7.4, we show how to guess the parameter $k$ based on $n$.) To this end, we introduce $k$-segments, which are subsets of node-cost sets. Each tuple of a segment represents a different embedding image with respect to the same subtree of the path tree. We then show how the five set operators must be adapted to find the best $k$ — or even all — second-level queries.

### 7.2.1 Representing Embedding Images

In Section 6.3, we introduced five set operators. A set consists of node-cost tuples, where each tuple $(u_D, c, \tilde{c})$ represents a data-tree node $u_D$ that matches a certain query selector. The primary cost $c$ assigned to $u_D$ reflects an intermediate embedding cost computed during the evaluation of the query-execution plan; the component $\tilde{c}$ serves as *backup cost* for the primary cost $c$. Now, we extend the tuples in such a way that they represent not only the cost of an embedding of a query subtree, but also the image of that embedding:

**Definition 7.5 (Extended node-cost tuple)** *An extended node-cost tuple is a structure* $(u_P, c, \tilde{c}, N)$*, where* $u_P$ *is a path-tree node,* $c$ *is a primary cost,* $\tilde{c}$ *is a backup cost, and* $N$ *is a set of extended node-cost tuples.*

The set $N$ contains $k$ extended node-cost tuples, where $k$ is the number of children of the query node $u_Q$ matching $u_P$. Each of these $k$ tuples represents a path-tree node that is a match of a child of $u_Q$. Thus, the whole extended node-cost tuple represents the image of an embedding with cost $c$.

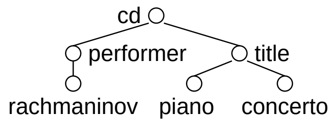

Figure 7.4: A query tree.

Consider the query tree depicted in Figure 7.4 and assume that the node with value cd matches the path-tree node $u_{P_1}$, performer matches $u_{P_2}$, rachmaninov matches $u_{P_3}$, title matches $u_{P_4}$, piano matches $u_{P_5}$, and concerto matches $u_{P_6}$. Then a particular tuple computed for the query tree has the form

$$(u_{P_1}, c_1, \tilde{c}_1, \{\, (u_{P_2}, c_2, \tilde{c}_2, \{\, (u_{P_3}, c_3, \tilde{c}_3, \emptyset) \,\}), (u_{P_4}, c_4, \tilde{c}_4, \{\, (u_{P_5}, c_5, \tilde{c}_5, \emptyset), (u_{P_6}, c_6, \tilde{c}_6, \emptyset) \,\}) \,\}),$$

where $c_1 \ldots c_6$ and $\tilde{c}_1 \ldots \tilde{c}_6$ are the primary and backup costs calculated during the evaluation of the query.

### 7.2.2 $k$-Segments

In Section 6.3, we defined node-cost tuples. A node-cost tuple refers to the root of a data subtree that includes the image of a query-tree embedding. In any set of node-cost tuples processed by the operators in a query-execution plan, each data-tree node is represented at most once. The unique occurrence of each node was reasonable because we were interested in the best embedding per matching subtree of the data tree, and therefore we kept only the lowest embedding cost per subtree. Now, we want to find the images of the best $k$ approximate embeddings of a query in a path tree in order to use them as second-level queries. Two distinct path-tree embeddings may have distinct embedding roots, for instance the cd node and the mc node. However, two distinct embeddings often have the same root, but map to different included subtrees of the path tree. For example, two embedding images may have the common root cd, but one of them may include the composer path whereas the other may include the performer path. To track the images of the best $k$ embeddings (and their costs) per query subtree and per subtree of the path tree, we use *$k$-segments*:

**Definition 7.6 ($k$-Segment)** *Let $k$ be a fixed number. A $k$-segment is a set $\{\, t_1, t_2, \ldots, t_l \,\}$ $(1 \leq l \leq k)$, where each $t_i = (u_{P_i}, c_i, \tilde{c}_i, N_i)$ $(1 \leq i \leq l)$ is an extended node-cost tuple. For each pair of integers $i, j$ $(1 \leq i < j \leq l)$ holds: $u_{P_i} = u_{P_j} \wedge (c_i \neq c_j \vee N_i \neq N_j)$.*

All tuples of a fixed $k$-segment represent embedding images of the *same* query subtree in the *same* subtree of the path tree.

### 7.2.3 Operators for Sets with Extended Node-Cost Tuples

We define the operators for sets with extended node-cost tuples. To distinguish these operators from the basic operators introduced in Section 6.3, we call them "$k$-operators". The interfaces of the $k$-operators join, outerjoin, union, and intersection get an additional parameter $k$. For the definition of the $k$-operators, we need an auxiliary function *prune*, which modifies a set in such a way that all segments are reduced to the $k$ lowest-cost tuples:

**Definition 7.7 ($k$-Set)** *Let $S$ be a set of node-cost tuples and $k$ be a fixed segment size. The $k$-set of $S$ is defined by the function*

$$\begin{aligned} \mathit{prune}(k, S) = \{\, & (u_P, c_1, \tilde{c}_1, N_1) \mid (u_P, c_1, \tilde{c}_1, N_1) \in S \\ & \wedge \text{ there exist at most } k-1 \text{ tuples } (u_P, c_2, \tilde{c}_2, N_2) \in S \text{ such that } c_2 < c_1 \,\}. \end{aligned}$$

## Selection

The selection operator for sets with extended node-cost tuples is almost identical to the corresponding basic operator (see Definition 6.5 on page 78), except that it initializes the additional set component of each tuple.

**Definition 7.8 (Selection)** *Let $\tau = (D, P)$ be a type, $\phi \in P$ be a predicate, $\alpha \in D$ be a value, $c_t$ be a transformation cost, and $T_P = (N_P, E_P, r_P, \mathit{type}_P, \mathit{value}_P)$ be a path tree. The selection of extended node-cost tuples from $T_P$ is defined as*

$$\sigma^{c_t}[\tau, \phi, \alpha]\ T_P = \{\, (u_P, c_t, c_t, \emptyset) \mid u_P \in N_P \wedge \tau \preceq \mathit{type}_P(u_P) \wedge \phi(\alpha, \mathit{value}_P(u_P)) \,\}.$$

## $k$-Join and $k$-Outerjoin

The basic join operator (see Definition 6.7 on page 79) selects all node-cost tuples from the first operand set that have descendants in the second operand set. For each ancestor and for each of its descendants, it calculates the sum of the primary cost of the ancestor, the primary cost of the descendant, and the node distance between the ancestor and the descendant. From all descendants of the same ancestor, the operator chooses the one with the lowest cost and inserts a new tuple consisting of the ancestor and the updated cost into the set of results. The transformation cost is added to the cost component of each result tuple. The $k$-join operator differs from the basic version in three details: First, it handles ancestor and descendant sets with $k$-segments. Second, it creates result segments of size $k$, where the tuples of a particular segment represent the images of distinct embeddings of the same query subtree into the same subtree of the path tree. Third, for each result tuple, it copies the set component of the ancestor tuple and extends it by the descendant tuple.

**Definition 7.9 ($k$-Join)** *Let $S_1$ and $S_2$ be sets of extended node-cost tuples, $c_t$ be a transformation cost, and $k$ be a segment size. The $k$-join between $S_1$ and $S_2$ is defined as*

$$\begin{aligned} S_1 \overset{k}{\bowtie}{}^{c_t} S_2 = \mathit{prune}(k, \{\, & (u_P, c_t + c_1 + c_2 + \mathit{nodedist}(u_P, v_P), \tilde{c}_1, N_1 \cup \{\, (v_P, c_2, \tilde{c}_2, N_2) \,\}) \mid \\ & (u_P, c_1, \tilde{c}_1, N_1) \in S_1 \wedge (v_P, c_2, \tilde{c}_2, N_2) \in S_2 \wedge u_P \rightsquigarrow v_P \,\}). \end{aligned}$$

The basic outerjoin operator (see Definition 6.8 on page 80) is similar to the basic join operator, except that it passes all node-cost tuples from the first operand set to the result set, whether they have descendants in the second operand set or not. Consequently, the definition of the $k$-outerjoin is almost identical to the definition of the $k$-join, except that each tuple of the ancestor set is inserted into the result set. The primary cost of the additional tuples is calculated as the sum of the transformation cost, the primary cost of the current ancestor tuple, and the deletion cost of the query leaf.

**Definition 7.10 ($k$-Outerjoin)** *Let $S_1$ and $S_2$ be sets of extended node-cost tuples, $c_t$ be a transformation cost, $c_d$ be a deletion cost, and $k$ be a segment size. The $k$-outerjoin between $S_1$ and $S_2$ is defined as*

$$\begin{aligned} S_1 \overset{k}{⟕}{}^{c_t}_{c_d} S_2 = \mathit{prune}(k, \{\, & (u_P, c_t + c_1 + c_2 + \mathit{nodedist}(u_P, v_P), \tilde{c}_1, N_1 \cup \{\, (v_P, c_2, \tilde{c}_2, N_2) \,\}) \mid \\ & (u_P, c_1, \tilde{c}_1, N_1) \in S_1 \wedge (v_P, c_2, \tilde{c}_2, N_2) \in S_2 \wedge u_P \rightsquigarrow v_P \,\} \cup \\ & \{\, (u_P, c + c_1 + c_d, \tilde{c}_1, N_1) \mid (u_P, c_1, \tilde{c}_1, N_1) \in S_1 \,\}). \end{aligned}$$

### $k$-Union and $k$-Intersection

The basic union operator (see Definition 6.9 on page 80) creates the union of the operand sets. If a node appears in only one operand set, then the cost calculated for the resulting node-cost tuple is the sum of its primary cost and the transformation cost passed to the operator. If a node is in both operand sets, then the minimum of the primary costs of the operands is chosen and increased by the transformation cost. The $k$-union operator extends that behavior to sets with $k$-segments. At most $k$ tuples representing the same path-tree node form a new segment, which is ensured by the *prune* function.

**Definition 7.11 ($k$-Union)** *Let $S_1$ and $S_2$ be sets of extended node-cost tuples, $c_t$ be a transformation cost, and $k$ be a segment size. The $k$-union between $S_1$ and $S_2$ is defined as*

$$S_1 \overset{k}{\sqcup}{}^{c_t} S_2 = \mathit{prune}(k, \{\, (u_P, c_t + c, \tilde{c}, N) \mid (u_P, c, \tilde{c}, N) \in S_1 \cup S_2 \,\}.$$

The basic intersection operator (see Definition 6.10 on page 81) selects all pairs of node-cost tuples from the operand sets that refer to the same data-tree node. For each pair, it creates a new node-cost tuple consisting of the shared node and a cost, which is the sum of the primary costs of the corresponding tuples, plus the transformation cost passed to the operator. The $k$-intersection operator differs from the basic version in two details: First, it passes $k$-segments of tuples to the result set. Second, it merges the set components of corresponding tuples.

**Definition 7.12 ($k$-Intersection)** *Let $S_1$ and $S_2$ be sets of extended node-cost tuples, $c_t$ be a transformation cost, and $k$ be a segment size. The $k$-intersection between $S_1$ and $S_2$ is defined as*

$$\begin{aligned} S_1 \overset{k}{\sqcap}{}^{c_t} S_2 = \mathit{prune}(k, \{\, & (u_P, c_t + c_1 + c_2 - \tilde{c}_1, \tilde{c}_1, N_1 \cup N_2) \mid \\ & (u_P, c_1, \tilde{c}_1, N_1) \in S_1 \wedge (u_P, c_2, \tilde{c}_2, N_2) \in S_2 \,\}. \end{aligned}$$

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

Having extended the five operators by the ability to track the embedding images, we can use the algorithmic framework developed in Chapter 6, which consists of the algorithms `create`$(Q)$ and `evaluate`$(\mathcal{P}, T_D)$. The former creates an execution plan $\mathcal{P}$ for the query $Q$; the latter evaluates $\mathcal{P}$ with respect to the data tree $T_D$. We now assume that the algorithm `create`$(Q)$ uses $k$-operators to assemble plans, and passes an additional parameter $k$ to the evaluation algorithm, which in turn passes it to the $k$-operators. If we set $k = \infty$, then we find *all* second-level queries in the path tree $T_P$ of $T_D$:

$$\texttt{evaluate}(\texttt{create}(Q), T_P, \infty).$$

To find the *best* $k$ second-level queries, we use a function $\mathsf{sort}(S, k)$ that sorts the second-level queries in $S$ by increasing costs, and selects the $k$ ones with the lowest costs:

$$\texttt{sort}(\texttt{evaluate}(\texttt{create}(Q), T_P, k), k).$$

The additional sorting is necessary because the $k$-operators track the best $k$ embedding images *per subtree* of the path tree. Among the embedding images for *all* subtrees, the best $k$ must be selected.

## 7.3 Finding Results of Second-Level Queries in a Data Tree

The query-evaluation algorithm proposed in Section 6.4 (which now uses $k$-operators) returns a set of cost-sorted tuples, where each tuple represents a second-level query. A second-level query can be evaluated by a simple algorithm that traverses the data tree and finds exact matches of the query. There is, however, a slight difference from the theoretical setting in Section 7.1: The algorithm `evaluate`$(\mathcal{P}, T_P, k)$ does not return "full" embedding images, but only "skeletons" of them. More precisely, it returns images that do not represent the inserted nodes, because the costs of the nodes to be inserted have been derived from the encoding of the path tree. Fortunately, it is not necessary to know the nodes implicitly inserted between two skeleton nodes $u_P$ and $v_P$ because a path tree preserves all parent-child relationships of its data tree. If $u_D, v_D$ are data-tree nodes and $[u_D], [v_D]$ are their node classes, then the path between $u_D$ and $v_D$ (if any) consists of nodes with the same types and values as the nodes on the path between $[u_D]$ and $[v_D]$. If we use any of the models to assign costs to basic transformations described in Section 5.4, then

$$\mathit{nodedist}(u_D, v_D) = \mathit{nodedist}([u_D], [v_D])$$

holds for each pair $u_D, v_D$ of data-tree nodes that have an ancestor-descendant relationship. To find the paths of the data-tree that are instances of the path between $u_P$ and $v_P$, we must select all instances of both nodes, and test whether they have ancestor-descendant relationships.

Algorithm 7.1 on the next page finds all exact embeddings of a second-level query (represented by the tuple $(u_P, c_1, \tilde{c}_1, N)$) into a data tree. The algorithm starts at the root of the second-level query, traverses the query tree top-down, and evaluates the results bottom-up. During its descent, it selects the node instances of the path-tree node $u_P$ in the data tree $T_D$ (Line 1). The node $u_P$ is the root of the currently processed subtree of the second-level query. Next, the algorithm evaluates each child $v_P$ of $u_P$ separately (Line 2): First, the subtree rooted at $v_P$ is evaluated and the embedding roots are stored in set $S_D$ (Line 3). Second, all tuples of $S_A$ that have descendants in $S_D$ are selected and inserted into a temporary set $S_T$ (Lines 5–7). At the end of the inner loop, $S_T$ is swapped to $S_A$, so that it now contains only tuples that have descendants in $S_D$. The same procedure is applied to each child of the current query-tree node, so that the returned set $S_A$ consists of all data-tree nodes that are roots of subtrees that include matches of the entire query subtree.

## 7.4 An Incremental Algorithm for the Best-$n$-Results Problem

So far, we have seen how to construct second-level queries and how to find the roots of all results for each second-level query. Algorithm 7.2 on the facing page integrates both parts. At Line 1, the best $k$ second-level queries are created and sorted by embedding cost. Next, the queries are evaluated by function `secondary`$(u_P, N_1, T_D)$. Each data-tree node retrieved by this function is the root of a logical document; the embedding cost of $Q$ for this document is determined by the tuple $(u_P, c, \tilde{c}, N)$ taken from $S_P$. Because any logical document may be matched by several second-level queries (with different embedding costs), the algorithm must keep track of documents already retrieved. Line 6 sketches the test against the history of the

**Algorithm 7.1** finds all results of a second-level query.

```
function secondary($u_P, N_1, T_D$)
params:  $u_P$ – a path-tree node representing the root of a second-level query,
         $N_1$ – a set of tuples representing the children of $u_P$,
         $T_D$ – a data tree,
returns: $S_A$ – the set root nodes of results for the second-level query.
1: $S_A := instances(u_P, T_D)$ /* select all node instances of $u_P$ in $T_D$ */
2: foreach $(v_P, c, \tilde{c}, N_2) \in N_1$ do
3:    $S_D :=$ secondary($v_P, N_2$) /* recursive call */
4:    $S_T := \emptyset$ /* initialize a temporary set */
5:    foreach $u_D \in S_A$ do
6:       if $u_D$ has a descendant in $S_D$ then
7:          $S_T := S_T \cup \{ u_D \}$
8:    $S_A := S_T$
9: return $S_A$
```

preorder numbers of the roots of retrieved documents. In practice, we use a hash table to implement this test. Algorithm 7.2 fulfills the requirements of an evaluation algorithm that solves the all-results problem (see Definition 5.12 on page 62).

**Algorithm 7.2** retrieves the results for the best $k$ second-level queries.

```
input:  $Q$   – an approXQL query,
        $T_D$ – a data tree,
        $T_P$ – the path tree of $T_D$,
        $k$   – the number of second-level queries to generate for $n$ results,
output: the root-cost pairs of the logical documents for the best $k$ second-level queries.
1: $S_P :=$ sort(evaluate(create($Q$), $T_P, k$), $k$)
2: foreach $(u_P, c, \tilde{c}, N) \in S_P$ do
3:    $S_D :=$ secondary($u_P, N, T_D$)
4:    foreach $u_D \in S_D$ do
5:       if $u_D$ has not been seen so far then
6:          output $(u_D, c)$
```

However, our main objective is to solve the more interesting best-$n$-results problem (see Definition 5.13 on page 62). If we knew beforehand how many second-level queries were necessary to find $n$ results, then we could determine $k$ from $n$ and pass it to Algorithm 7.2. Unfortunately, there is no strong correlation between $k$ and $n$; some second-level queries may retrieve many results, some may not find any result at all. Therefore, we must *guess* the initial value of $k$ and increment it by a delta value $\delta$ if the first $k$ second-level queries have not retrieved enough results. Fortunately, the increment of $k$ does not invalidate the previous results: The set $S_P$ returned by algorithm `evaluate` for a certain $k$ is a subset of the set $S'_P$

returned for a $k' \geq k$.

Algorithm 7.3 shows an incremental version of Algorithm 7.2. After the construction of the execution plan for $Q$ (Line 1), the algorithm repeats the creation and evaluation of second-level queries until $n$ results have been found. Note that the algorithm terminates even if $n$ is infinite, because no second-level query is created twice and because the number of included trees in the path tree is finite. In any step, the algorithm erases the prefix of all second-level queries that have already been evaluated (Line 5) and adds $\delta$ to $k$ (Line 7). All second-level queries remaining in $S_P$ are evaluated as described for Algorithm 7.2.

**Algorithm 7.3** retrieves incrementally the best $n$ results of a query.

```
input:  $Q$   – a query,
        $T_D$ – a data tree,
        $T_P$ – the path tree of $T_D$,
        $k$   – initial guess for the number of second-level queries to create,
        $n$   – the number of requested results,
        $\delta$   – increment for $k$ if the number of queries is not sufficient,
output: the root-cost pairs of the best $n$ results for $Q$ in sorted order.
1: $\mathcal{P}$ := create($Q$) /* create the execution plan for $Q$ */
2: $k_{prev} := 0$
3: while the number of retrieved documents is less than $n$ do
4:     $S_P$ := sort(evaluate($\mathcal{P}, T_P, k$), $k$)
5:     remove the $k_{prev}$ lowest-cost entries from $S_P$
6:     execute Lines 2 – 6 of Algorithm 7.2
7:     $k_{prev} := k$; $k := k + \delta$
```

What are appropriate values for $k$ and $\delta$? In our experiments (see Chapter 10), we could not find an initial value for $k$ that was satisfying for all queries and data trees. This is not surprising because all horizontal dependencies of the data tree are lost in the path tree, and therefore we cannot say beforehand whether two query siblings have matches in the same subtree. In most cases, we got the fastest answers if we set $k = \max(200, 3 \cdot n)$. Also, it proved to be successful if we incremented $k$ non-linearly. We set $\delta = k$ and doubled the value of $\delta$ in each iteration step.

## 7.5 Optimizing Schema-Driven Query Evaluation

The schema-driven incremental query evaluation proposed in this chapter guarantees that we get the best $n$ answers without computing the whole set of results. However, our approach still has a weakness: The selectivity of a second-level query is typically high, and therefore the algorithm must often choose a large $k$ to find $n$ results. In this section, we propose a *hybrid* query-evaluation method, which merges second-level queries with similar substructures. This reduces the selectivities so that fewer queries must be created in order to find $n$ results.

To motivate our approach, consider the query

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

and assume that we are allowed to change `title` into `category` or `description`, `composer` into `performer`, and the word `concerto` into `sonata` or `symphony`. Also assume that all parent-child relationships appearing in the query existed at least once in a path tree. Figure 7.5 shows a detail of a path tree for which our assumption holds. If no deletions of structural nodes were permitted, then Algorithm 7.3 would create 26 distinct second-level queries (provided that $k \geq 26$). Many of them might find no results.

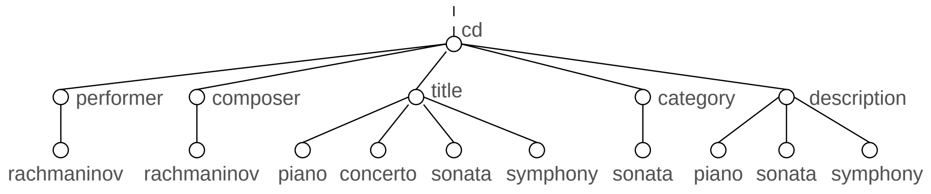

Figure 7.5: A detail of a path tree.

To reduce the selectivities of second-level queries, we merge them during the creation phase. Merging candidates are queries that have subtrees with

- different structures but equal embedding costs,
- identical structures but different leaf values and different embedding costs.

Subtrees with different structures but equal embedding costs particularly appear if the value-change costs are equal for several alternative values of an inner query selector. In our example, we may define that the change of `title` to `category`, and of `title` to `description` both have cost 4. Then we can reduce the number of second-level queries from 26 to 24, which also decreases the average selectivity per query. Of course, if many alternative values with equal costs are specified, the reduction is much more effective.

The motivation behind merging second-level queries that have subtrees with identical structures but different leaf values is the observation that leaves — in contrast to inner nodes — typically have very high selectivities. In fact, the leaves primarily determine the selectivity of a query. A second-level query in which each leaf is annotated with all alternative values selects many more results than a query with a single value per selector. Moreover, the evaluation time of a merged second-level query is not significantly higher than that of the original second-level query, because the union of the match sets for all alternative leaf values is small and can be constructed efficiently.

In most cases, a large number of second-level queries can be merged. In our example, there are in fact only six second-level queries with a distinct inner structure. If we additionally assume equal value-change costs for both alternative values of `title`, then we can reduce the number of second-level queries to four (see Figure 7.6).

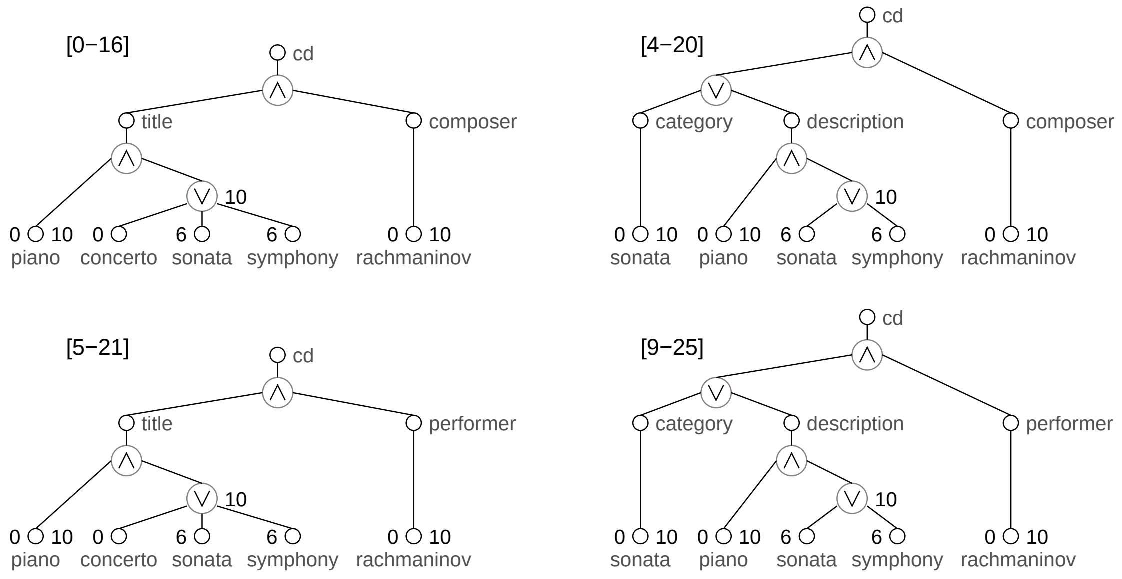

Figure 7.6: The four hybrid second-level queries created for the approXQL query `cd[title["piano" and "concerto"] and composer/"rachmaninov"]` with respect to the path-tree shown in Figure 7.5 on the page before. The deletion of inner nodes is not permitted; the deletion of a leaf imposes cost 10. The following value changes are defined: composer to performer with cost 5, title to category or description with cost 4, concerto to sonata or symphony with cost 6. The intervals show the minimum and maximum costs the query results can have.

As the figure shows, we construct merged second-level queries by connecting alternative subtrees with "$\vee$"-nodes. Sibling nodes in the original second-level query are connected by "$\wedge$"-nodes. Because alternative leaves may impose distinct costs, we annotate each leaf with its value-change cost (left side). Furthermore, each leaf whose deletion cost is less than infinite is annotated with this deletion cost. If a leaf only has a single value, then we write the deletion cost at the right side of the node. Otherwise, we annotate the "$\vee$"-node that connects the set of nodes with alternative values. A second-level query with conjunctive and disjunctive parts is called a *hybrid second-level query*.

The semantics of the cost annotations is as follows: A second-level query has a minimum embedding cost, which is determined at creation time (see Section 7.2). During the evaluation of the query, an extended version of Algorithm 7.1 on page 121 evaluates all alternative branches of the query. The cost of a result is calculated as the cost assigned to the second-level query, plus the sum of the costs assigned to the lowest-cost combination of leaves that have matches in the result.

Consider the lower-left query shown in Figure 7.6 on the preceding page. The query has the minimum embedding cost 5 because the value `composer` has been changed to `performer`. Assume that during the evaluation of the query the algorithm detects that no match of the leaf `piano` exists for a particular subtree of the data tree, and that only a match for the leaf `sonata` but not for `concerto` exists. If the deletion has cost 10 and the value change has cost 6, then the algorithm adds 16 to the embedding cost of the query and assigns the cost 21 to the result. This cost is the maximum cost a result of this query can have. The duplicate cost calculations, first during the creation of second-level queries (as for the schema-driven approach), and second during their evaluation (as for the direct approach), are the reasons for the name *hybrid* query-evaluation method.

The cost intervals imply that the sort order of second-level queries does not fully determine the sort order of their results: The cost assigned by the query-creation algorithm only determines the *minimum* cost of a result selected by the query. The actual cost may be higher if a leaf must be deleted or if its value must be changed. In this case, the subsequent second-level query (with a higher embedding cost) may find a result with a lower cost than the previous one. It follows that Algorithm 7.2 on page 121 (and its incremental version) does not work correctly for hybrid second-level queries — it cannot output the results of a second-level query without checking the embedding cost of the subsequent query. We adapt Algorithm 7.2 to work with hybrid second-level queries by adding a buffer for "unsafe" results. All results returned by a second-level query are inserted into the buffer in cost-sorted order. Before a new second-level query is evaluated, the algorithm verifies whether there are results in the buffer whose costs are less than or equal to the embedding cost of that query. If such buffer entries exist, they are output in sorted order and removed from the buffer.

In Chapter 10, we investigate the efficiency of the hybrid query-evaluation method, and compare it with both the direct method proposed in the previous chapter and the schema-driven approach that does not merge second-level queries.

## 7.6 Related Work

The efficient retrieval of the best $n$ results is a challenge for the implementation of any system that in some sense valuates the results of a query. Efficient solutions have been proposed for text retrieval systems (see [Bro95] for an overview) and, more recently, for relational database management systems (see, e.g., [CK97, CG99, NB02]). However, in the context of querying XML data, little attention has been paid to the problem.

Amer-Yahia et. al. [ACS02] investigate two problems related to the evaluation of relaxed tree-pattern queries: the threshold problem and the best-$n$ problem (called top-$k$ problem in their framework). A threshold is a user-defined value that determines the maximum score a result may have. The authors propose two techniques to solve the threshold problem: First, relaxations encoded in a query-execution plan are "undone" if those relaxation would lead to results with scores above the threshold. Second, intermediate results are discarded if their scores are above the threshold. The best-$n$ problem is treated as a dynamic variant of the threshold problem, where the threshold is adapted after the evaluation of each operator in the plan. To each operator, a number `maxW` is assigned, which indicates the maximum score by which the score of an intermediate results can grow in all remaining evaluation steps. The score of the $n$th intermediate result determines the threshold. Each result whose intermediate score augmented by `maxW` is below the current threshold is discarded. An inherent weakness of that algorithm is that it compares maximum final scores with intermediate scores so that it can discard only few results at early evaluation steps. In fact, the authors point out that the threshold algorithm always performed better in their experiments provided that $n$ is accurately estimated. However, they do not provide a method to estimate this crucial parameter.

A structural summary of a semistructured database helps to explore the logical structure of the database and to accelerate the evaluation of queries with (regular) path expressions. Examples of such structural summaries are DataGuides [GW97] and T-indexes [MS99]. To our knowledge, the use of a structural summary for the retrieval of the best $n$ results has not been considered by other authors.