Chapter 6

Direct Query Evaluation

The approximate query-matching model proposed in the previous chapter builds on the concept of a query closure, which contains all query trees that can be derived from a user's query via transformation sequences. All query trees are embedded into the data tree, and for each logical document, the query tree with the lowest embedding cost determines the score. Unfortunately, this naive query-evaluation method cannot be used in practice, because we can create an infinite number of query trees only by inserting nodes.

In this chapter, we go beyond the naive approach and present an integrated, modular, and performant algorithmic framework to solve the all-results problem and the best-n-results problem (see Definitions 5.12 and 5.13 on page 62). *Integrated* means that the evaluation of a query is based on a single query-execution plan. *Modular* means that arbitrary query-execution plans can be assembled from a small set of operators. *Performant* means that all results of a query can be found in polynomial — typically sublinear — time with respect to the number of nodes in the data tree. The framework introduced in this chapter is in fact more general: It can be used to find the images of the best query-tree embeddings into a schema, which are then used as second-level queries to be executed against the data tree (see Chapter 7).

The chapter is structured as follows: In the first section, we provide an overview of our query-evaluation method. In Section 6.2, we introduce the expanded representation of a query, which is a compact form of the query closure. The expanded representation is the starting point for the construction of a query-execution plan. In Section 6.3, we define five cost-calculating set operators; in Section 6.4, we show how they can be assembled to query-execution plans. We prove the completeness and soundness of our approach in Section 6.5, and discuss techniques to optimize the evaluation of a plan in Section 6.6. In Section 6.7, we analyze the upper

bounds for the space complexity of query-execution plans. Finally, in Section 6.8, we review related work.

6.1 From the Semantics of a Query to its Evaluation: An Overview

The objectives of our query-evaluation method are twofold: First, we want to avoid the explicit creation of query closures. Second, we want to overcome the principle that all transformed query trees are independently embedded into the data tree.

To meet the first objective, we shift from the closure of a query to its *expanded representation*, which is a tree that represents all query trees in the closure. To construct the expanded representation, we make use of the fact that transformation sequences are ordered, and particularly that insertions are the last operations in a sequence (see Definition 5.8 on page 61). Because of this order, we can postpone the insertion of nodes, and create a tree that explicitly encodes all permitted deletions, permutations, and value changes; but also implicitly represents all permitted node insertions by means of "expandable" edges.

To meet the second objective, we propose *query-execution plans*. The structure of a plan closely resembles the structure of the expanded query representation for which it is created. However, all nodes in the expanded representation are replaced by operators: The selection operator returns all data-tree nodes of a given type and value. The union and intersection operators essentially perform standard operations on sets of data-tree nodes, but additionally apply cost functions to the operand pairs. The join operator implements the evaluation of "expandable" edges. It verifies whether pairs of data-tree nodes are connected by paths, and calculates the "node distances" between them. The node distance is the total cost of all nodes that must be inserted between two query-tree nodes in order to map the query-tree path to a particular data-tree path.

During the evaluation of a query-execution plan, the lowest-cost embeddings for all logical documents are computed in parallel. The structure of a plan implies a greedy strategy to determine the lowest-cost embedding per document: Starting at the bottom-most plan operators and at the leaves of the documents, the optimal solutions (the lowest-cost embeddings) for all subplans and subdocuments are computed. The optimal solutions for the subproblems are then combined in the next step, in order to find the optimal solutions for a larger subproblem. The result of the evaluation of a plan is a set of node-cost tuples, where each tuple represents the root of a logical document and the cost of the lowest-cost query em-

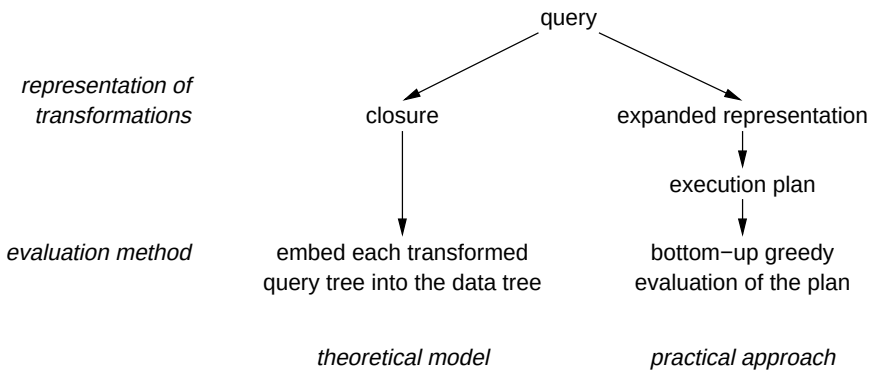


Figure 6.1: Relationships between the semantics of approXQL and the evaluation of a query.

bedding. Figure 6.1 summarizes the relationships between the approXQL semantics and the query-evaluation method proposed in this chapter.

6.2 The Expanded Representation of a Query

An expanded query representation is a tree that represents the closure of a query. It explicitly encodes all deletions, permutations, and value changes, and implicitly represents all insertions. We first give the definition of an expanded query representation, and then introduce the rules that map a query to its expanded representation.

Definition 6.1 (Expanded query representation) *An expanded query representation is a rooted tree* $T_E = (N, E, r, op, type, value, pred, mod, valchg, delcost, transcost)$*, where* N *is a set of nodes,* $E \subseteq (N \times N)$ *is a set of edges,* r *is the root of* T_E*, and*

$$op : N \to \{ s, \wedge, \vee \}, \quad value : N \to D^*, \quad mod : N \to M, \quad delcost : N \to \mathbb{R}^+,$$
$$type : N \to \mathcal{T}, \quad pred : N \to P^*, \quad valchg : N \to 2^{D^* \times \mathbb{R}^+}, \quad transcost : E \to \mathbb{R}^+$$

are functions, where $\mathcal{T}$ *is a finite set of types,* $P^* = \bigcup_{(D,P)\in\mathcal{T}} P$ *is the union of predicates in* $\mathcal{T}$*,* $D^* = \bigcup_{(D,P)\in\mathcal{T}} D$ *is the union of domains in* $\mathcal{T}$*,* $M = \{ \mathsf{insres}, \mathsf{insrel}, \mathsf{delres}, \mathsf{valres} \}$ *is a set of modifiers, and* $\mathbb{R}^+$ *is the set of non-negative real numbers. All nodes* $u_E \in N$ *with* $op(u_E) = s$ *have at most one child; all other nodes in* N *have two children. The functions type, value, pred, mod, valchg, and delcost are defined only for nodes* $u_E \in N$ *with* $op(u_E) = s$*.*

We call a node $u_E \in N$ with $op(u_E) = s$ an *s-node* (selection node), with $op(u_E) = \wedge$ a $\wedge$*-node* (conjunction node), and with $op(u_E) = \vee$ a $\vee$*-node* (disjunction node). The nodes have the following semantics:

s-nodes represent selectors of the original query. An *s*-node u_E has the same type, value, predicate, and modifiers as the corresponding query selector (functions $type(u_E)$, $value(u_E)$, $pred(u_E)$, and $mod(u_E)$). It is annotated with a set of value-cost pairs (function $valchg(u_E)$) consisting of all alternative values for $value(u_E)$ together with the value-change costs. The function $delcost(u_E)$ returns the deletion cost of the query selector represented by u_E.

$\wedge$-nodes represent "`and`" operators in the original query.

$\vee$-nodes have two functions: First, they represent "`or`" operators in the query. Second, they represent deletions and permutations. In the latter case, one operand of a $\vee$-node u_E is the root of the original subquery, and the other is the root v_E of the modified subquery. The edge leading to the second operand is annotated with the deletion or permutation cost, respectively (function $transcost(u_E, v_E)$).

The children of a node are ordered. We write $child(u_E)$ to refer to the child of an *s*-node u_E, and $child_1(u_E)$, $child_2(u_E)$ to refer to the two operand nodes of a $\wedge$-node or a $\vee$-node u_E. A node v_E is called an *s-child* of u_E if v_E is an *s*-node and there is a path from u_E to v_E such that all other nodes on that path are $\wedge$-nodes or $\vee$-nodes. Then, u_E is the *s-parent* of v_E.

Figure 6.2 on the next page shows an example of an expanded query representation. For simplicity, we do not show the encoding of permutations. We also omit all types and predicates, all zero costs, and all empty modifier sets. The top-level node represents the `cd` selector of the original query and its alternative values `dvd` and `mc`, which have the costs 6 and 4, respectively. In this example, each $\vee$-node represents an alternative between keeping and deleting an inner node. For instance, the edge to the right operand of the left $\vee$-node leads to a subtree without a `title` node. It is annotated with the deletion cost 5 defined for `title` nodes. All leaves may be deleted, and are therefore annotated with deletion costs (8 for `piano`, 6 for `concerto`, and 11 for `rachmaninov`).

An edge leading to an *s*-node can be "relaxed": It represents an arbitrary number of nodes that can be inserted between the *s*-node and its *s*-parent. The actual number of inserted nodes is implicit; we show in Chapter 8 how the necessary number of insertions can be determined from the data tree. The function *mod* controls the relaxation: If the returned set contains the symbol insres, then insertions are forbidden. This symbol is returned for all nodes representing query selectors for which insert restrictions are defined (see Section 3.2.1). If insrel is in the returned set, then an insert relaxation is specified, which means that insertions impose no costs.

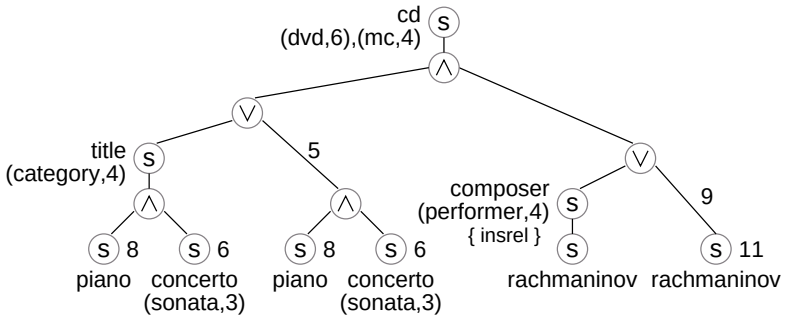


Figure 6.2: Expanded representation for the query `cd[title["piano" and "concerto"] and *composer/"rachmaninov"]`, where the following transformations are allowed: deletions of title (cost 5), composer (cost 9), piano (cost 8), concerto (cost 6), and rachmaninov (cost 11); value changes of cd to dvd (cost 6) or mc (cost 4), title to category (cost 4), composer to performer (cost 4), and concerto to sonata (cost 3); arbitrary insertions.

Each query tree in the closure of a query can be derived from the expanded representation by traversing the representation top-down and by accumulating the costs. Figure 6.3 depicts four query trees included in the expanded query representation shown in Figure 6.2. A number assigned to a node represents the embedding cost of the subtree rooted at this node. The left query tree can be derived from the expanded query representation as follows: Initialize a cost accumulator c. Start at the top-level node in the expanded query representation. Choose the value dvd with cost 6. Choose the left child of the top-level $\wedge$-node and follow the left child of the subsequent $\vee$-node to the `title` node. Add the value-change cost 4 of category to c. Proceed to the left child of the lower-left $\wedge$-node and add the deletion cost 8 to c. Proceed to the right child of the $\wedge$-node and add the value-change cost 3 belonging to sonata to c. Continue with the right child of the top-level $\wedge$-node and follow the right child of the $\vee$-node. Add the deletion cost 9 to c. Insert a node with type *struct* and value review to the query tree, and add the insertion cost 1 to c. The embedding cost of the query tree is 31 $(= 6 + 4 + 8 + 3 + 9 + 1)$.

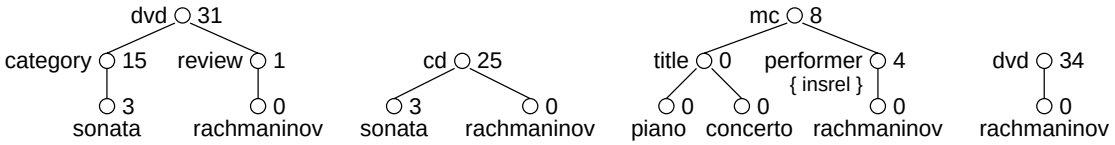


Figure 6.3: Examples of transformed query trees included in the expanded query representation shown in Figure 6.2.

In the following subsections, we describe the construction of an expanded query representation in detail. We first describe the mapping of a query to the "basic form" of its expanded

representation. Next, we show the encoding of transformations following the order

$$\text{deletions} \rightarrow \text{permutations} \rightarrow \text{value changes.}$$

Because the encoding of transformations follows the order in which transformations may be applied according to Definition 5.8 on page 61, it is guaranteed that all trees in the closure can be derived from the expanded representation. Note that all encoding steps can be integrated in a single algorithm. We separate the steps to make the construction process explicit.

6.2.1 Constructing the Basic Form of an Expanded Query Representation

The basic form of an expanded query representation models both the hierarchical relationships in the query and the logical relationships between subqueries at the same hierarchy level. Because the inter-subquery relationships may be specified by arbitrary complex expressions, we adapt the well-known concept of a *parse tree* [ASU86] to define the *operator tree* of an expression:

Definition 6.2 (Operator tree) *Let $F = expr(Q_1, Q_2, \ldots, Q_m)$ be a Boolean formula of queries. Let T be the parse tree of F, constructed by treating the queries $Q_1, Q_2, \ldots, Q_m$ as literals. The operator tree of F is derived from T by substituting each "*`and`*" by a $\wedge$-node, each "*`or`*" by a $\vee$-node, and each Q_i literal by a placeholder s-node.*

Using this definition, Algorithm 6.1 on the next page creates the basic form of an expanded query representation. The algorithm visits the subqueries of a query top-down. It creates an s-node for each selector, and uses it as the root of the operator tree created for the expression of immediate subqueries. We use a simplified notation for the algorithm: T_E is empty (Line 1) if its node and edge sets are empty, and if its functions are undefined. Defining $mod(u_E)$ (Line 5) means adding a mapping of u_E to the subset of M that reflects all restrictions/relaxations defined for the selector s. Adding a subtree T to u_E (Line 7) means merging the node and edge sets of T_E and T, and adding an edge from u_E to the root of T. Similarly, substituting a placeholder node v_E by T_{E_i} (Line 10) means substituting v_E by the root of T_E, and merging the sets and functions of T_E and T_{E_i}.

6.2.2 Encoding Deletions

A deletion of a node v_E is encoded into an expanded query representation by duplicating the subtree rooted at the child of v_E, and connecting the alternative subtrees by a $\vee$-node.

Algorithm 6.1 creates the basic form of an expanded query representation.

function `create_basic_representation`(Q)

params: Q – a query,
returns: the basic form of the expanded representation T_E of Q.

1: Let s be the root selector of Q
2: Create an empty expanded query representation T_E
3: Create an s-node u_E and add it as root to T_E
4: Define $type(u_E)$, $value(u_E)$, and $pred(u_E)$ according to the characteristics of s
5: Define $mod(u_E)$ according to the restrictions/relaxations defined for s
6: **if** $Q = s[expr(Q_1, Q_2, \ldots, Q_m)]$ **then** /* Q *has subqueries* */
7: Create the operator tree of $expr(Q_1, Q_2, \ldots, Q_m)$ and add it as subtree to u_E
8: **for** $i := 1$ **to** m **do**
9: T_{E_i} := `create_basic_representation`(Q_i)
10: Substitute the placeholder s-node for Q_i by T_{E_i}
11: **return** T_E

Algorithm 6.2 on the following page performs the encoding of all permitted deletions. Its input is the basic form of an expanded query representation T_E constructed by Algorithm 6.1 and a function *costs* that defines the deletion costs for the s-nodes in T_E. This function implements one of the cost-assignment methods discussed in Section 5.4. The algorithm iterates through all s-nodes for which the deletion is permitted (Line 1). For each inner node, it creates a $\vee$-node that connects the node to delete with a copy of the subtree rooted at its child (Lines 3–6). "Create a $\vee$-node x_E" means adding a new node to the set N of T_E. "Create a copy $T'_E[w'_E]$ of $T_E[w_E]$" means adding nodes to N and edges to E such that together they have the same structure as $T_E[u_E]$. It also means that the functions of T_E are extended such that they return the same values for the corresponding nodes and edges in $T'_E[w'_E]$ and $T_E[w_E]$. The deletion cost is assigned to the edge between the $\vee$-node and the child of the node being deleted (Line 7). If the current node is a leaf, then the algorithm simply assigns the deletion cost to that node (Line 9).

6.2.3 Encoding Permutations

A deletion is a simple modification of a query tree that concerns a single node. A permutation, in contrast, does not only involve two nodes, but also changes the structures of the subtrees rooted at the permuted nodes. The encoding of a permutation into an expanded query representation is therefore slightly more complicated. We first discuss which parts of the expanded representation are affected by a permutation, afterwards, we present the algorithm that encodes permutations.

Algorithm 6.2 encodes deletions into an expanded query representation.

input: $T_E = (N, E, r, op, type, value, pred, mod, valchg, delcost, transcost)$,
$costs : N \to \mathbb{R}^+$ – a cost-assignment function,
output: T_E with encoded deletions.

1: **for each** s-node $v_E \in N$ such that $v_E \neq r$ and delres $\notin mod(v_E)$ **do**
2: **if** v_E has a child **then**
3: Let u_E be the parent and w_E be the child of v_E
4: Create a copy $T'_E[w'_E]$ of $T_E[w_E]$
5: Create a $\vee$-node x_E
6: $E := (E \setminus \{ (u_E, v_E \}) \cup \{ (u_E, x_E), (x_E, v_E), (x_E, w'_E) \}$
7: $transcost(x_E, w'_E) := costs(v_E)$
8: **else** /* v_E *is a leaf* */
9: $delcost(v_E) := costs(v_E)$

A permutation is defined for pairs of query-tree nodes that have a parent-child relationship and have certain types and values. Even if two parent-child query selectors have the desired values, the permutation is applicable only for some of the query trees in the closure of the query: It is possible that one or both nodes to permute do not appear in the tree, because they are part of another "`or`" branch in the query, or because one or both nodes have been deleted. Consider the query

```
cd[title/"piano" and (composer/"rachmaninov" or performer/"ashkenazy")],
```

and assume that `cd` and `composer` nodes may be permuted with cost 9, but no other transformations are defined. Figure 6.4 shows the basic form of the expanded representation of the query. The two query trees depicted in Figure 6.5 can be derived from the expanded representation. Clearly, the permutation can only be applied to the upper tree, yielding the transformed tree shown in Figure 6.6.

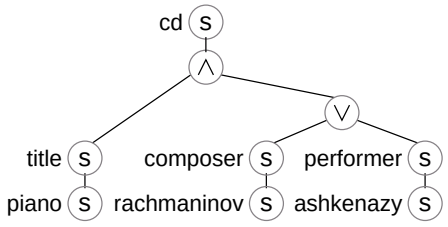


Figure 6.4: The basic form of an expanded query representation.

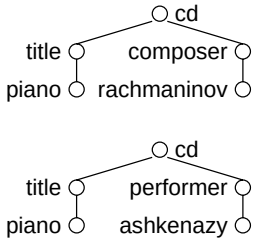


Figure 6.5: Two included query trees.

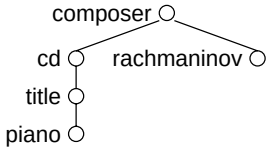


Figure 6.6: A permuted query tree.

How can a permuted tree be directly derived from an expanded representation? The path `composer-rachmaninov` shown in Figure 6.6 can be instantiated from the corresponding path

in the expanded representation shown in Figure 6.4. The subtree rooted at the `cd` node in Figure 6.6 can be derived from the `cd` subtree of the expanded representation by ignoring (i) the `composer` subtree (which gets a new position in the permuted tree) and (ii) the `performer` subtree (which is an alternative to the `composer` subtree and is not involved in the permutation).

The rule used in the example can be generalized in a straightforward way: Let u_E and v_E be two s-nodes in an expanded representation such that v_E is an s-child of u_E. Assume that u_E and v_E are allowed to be permuted. If we want to construct all query subtrees rooted at a node derived from u_E, but without a subtree derived from v_E, then we have to ignore (i) the subtree rooted at v_E, and (ii) all subtrees rooted at $\vee$-nodes on the path between u_E and v_E.

Algorithm 6.3 on the following page makes use of this rule to encode all allowed permutations into an expanded query representation. Its input is an expanded query representation T_E constructed by Algorithm 6.2, and the function *costs* that defines the permutation costs for pairs of s-nodes in T_E. The block within the outer loop at Line 1 consists of two parts: In the first part (Lines 2–13), the algorithm creates the standard structure for representing a permutation. In the second part (Lines 14–21), it inspects the operators on the path between the permuted nodes, and creates edges connecting all subtrees that represent parts of permuted subtrees. In the following, we use an example to explain the algorithm in more detail.

Consider the basic form of an expanded query representation shown in Figure 6.4 on the preceding page, and assume that the permutation of cd and composer with cost 9 is permitted. This permutation is selected at Line 1 of Algorithm 6.3 and encoded at Lines 2–21. The result of the encoding is the expanded representation depicted in Figure 6.7 on page 77. The nodes in the figure are annotated with the symbols used in the algorithm. We say that u_E is the upper node of the permutation, and v_E is the lower node. First, the algorithm makes copies of the nodes to be permuted (Line 2). Next, it creates a $\vee$-node x_E, which models the alternative between the original subtree and the subtree representing the permuted parts of the query trees (Line 3). Having created the three nodes, the algorithm adds edges between them and the original tree (Line 4). The edge from the $\vee$-node x_E to the alternative subtree is annotated with the permutation cost 9 (Line 5). If the upper node u_E has a parent p_E (not the case in our example), then the algorithm replaces the edge from p_E to u_E by an edge to the created $\vee$-node (Lines 6 and 7). At Lines 8–13, the algorithm connects the copies of the permuted nodes in reversed order. If the lower node v_E has a subtree, then it is duplicated, and an additional $\wedge$-node is necessary to add the copy of the subtree to v'_E. At Lines 14–21, the algorithm follows the path from the upper node u_E to the lower node v_E. Each $\wedge$-node

Algorithm 6.3 encodes permutations into an expanded query representation.

input: $T_E = (N, E, r, op, type, value, pred, mod, valchg, delcost, transcost)$,
$costs : N \times N \to \mathbb{R}^+$ – a cost-assignment function,
output: T_E with encoded permutations.

1: **for each** pair of s-nodes $u_E, v_E \in N$ for which a permutation is defined **do**
2: Create a copy u'_E of u_E and a copy v'_E of v_E
3: Create a $\vee$-node
4: $E := E \cup \{ (x_E, u_E), (x_E, v'_E) \}$
5: $transcost(x_E, v'_E) := costs(u_E, v_E)$
6: **if** u_E has a parent p_E **then**
7: $E := (E \setminus \{ (p_E, u_E) \}) \cup \{ (p_E, x_E) \}$
8: **if** v_E has a child w_E **then**
9: Create a copy $T'_E[w'_E]$ of $T_E[w_E]$
10: Create a $\wedge$-node y_E
11: $E := E \cup \{ (v'_E, y_E), (y_E, w'_E), (y_E, u'_E) \}$
12: **else** /* v_E *is a leaf* */
13: $E := E \cup \{ (v'_E, u'_E) \}$
14: **for each** $\wedge$-node q_E on the path from u_E to v_E **do**
15: Let r_E be the child of q_E not on the traversed path
16: Create a copy $T'_E[r'_E]$ of $T_E[r_E]$
17: **if** u'_E has a child s'_E **then**
18: Create a $\wedge$-node z_E
19: $E := (E \setminus \{ (u'_E, s'_E) \}) \cup \{ (u'_E, z_E), (z_E, r'_E), (z_E, s'_E) \}$
20: **else**
21: $E := E \cup \{ (u'_E, r'_E) \}$

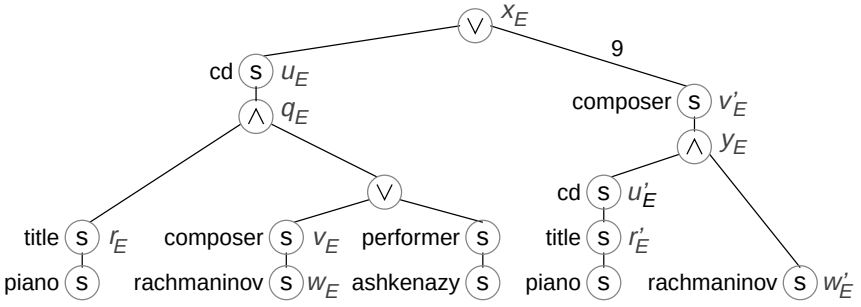


Figure 6.7: The expanded query representation depicted in Figure 6.4 on page 74 with the encoded permutation of cd and composer. The symbols assigned to the nodes refer to Algorithm 6.3 on the facing page.

on this path connects subtrees that represent parts of permuted query trees. Copies of these subtrees must be added to the copy u'_E of the upper node. In our example, the algorithm visits the $\wedge$-node q_E. The title path starting at r_E is part of all permuted trees and must be added to u'_E. If more than one $\wedge$-node exists (this is not the case in the example), the algorithm creates $\wedge$-nodes to connect the edges to the subtrees (Lines 19–21).

6.2.4 Encoding Value Changes

The encoding of value changes is the last step in the building process of an expanded query representation. For each s-node u_E of the tree constructed by Algorithm 6.3, the function $valchg(u_E)$ is initialized with a set of value-cost pairs that represent all possible value changes of the node. We omit the algorithm for this trivial operation.

6.3 Cost-Calculating Set Operations

We introduce five operators that are the basic building blocks of query-execution plans. With these operators, we establish the connection between the expanded representation of a query (as proposed in the previous section) and an execution plan for that query, which is the topic of the next section.

6.3.1 Node-Cost Tuples

All operators process sets of data-tree nodes and calculate (partial) embedding costs. To relate data-tree nodes to embedding costs, we define *node-cost tuples*:

Definition 6.3 (Node-cost tuple) *A node-cost tuple is a structure $(u_D, c, \tilde{c})$, where u_D is a data-tree node and $c, \tilde{c}$ are costs (non-negative real numbers).*

We say that c is the *primary cost* of u_D and $\tilde{c}$ is the *backup cost* of c. The node u_D is the match of a certain query selector; c is an intermediate cost computed during the evaluation of a query-execution plan. After the evaluation of the plan is completed, each node-cost tuple $(u_D, c, \tilde{c})$ in the result set represents the root u_D of a result of the query and the *approximate query-matching distance* c between the query and this result (see Definition 5.11 on page 62). The backup cost is necessary for the correct cost calculation in intersection operators (see below). To access the nodes in a set of node-cost tuples, we define the *node set* of a set of node-cost tuples:

Definition 6.4 (Node set) *Let S be a set of node-cost tuples. Its node set is defined as*

$$nodes(S) = \{\, u_D \mid (u_D, c, \tilde{c}) \in S \,\}.$$

6.3.2 Selection

The selection operator $\sigma^{c_t}[\tau, \phi, \alpha]\ T_D$ creates a new set of node-cost tuples consisting of all nodes in T_D that have type τ or a subtype of τ, and that fulfill the selection predicate ϕ with respect to value α. The transformation cost c_t is an additional argument not only for the selection, but also for all other operators. It is used to pass the cost of a deletion, a permutation, or a value change to the operator. If c_t is zero, then we omit it in both the textual and graphical representations of an operator. In the case of the selection operator, c_t initializes both the primary and backup costs of the created tuple.

Definition 6.5 (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 $T_D = (N_D, E_D, r_D, type_D, value_D)$ be a data tree. The selection of node-cost tuples from T_D is defined as*

$$\sigma^{c_t}[\tau, \phi, \alpha]\ T_D = \{\, (u_D, c_t, c_t) \mid u_D \in N_D \wedge \tau \preceq type_D(u_D) \wedge \phi(\alpha, value_D(u_D)) \,\}.$$

Consider a selection with the parameters $\tau = integer = (\mathbb{I}, \{=, <, \leq, >, \geq\})$, $\phi = \leq$, and $\alpha = 2000$. A data-tree node u_D is inserted into the result set if $(integer \preceq type_D(u_D)) \wedge (2000 \leq value_D(u_D))$ holds.

We use a simplified representation for selections in graphical query-execution plans if the type is clear from the context, and if the predicate tests for equality. For example, the operator σ title symbolizes the selection of a set representing all nodes that have type *struct* and whose value is equal to title.

6.3.3 Join and Outerjoin

The join operator $S_1 \mathbin{⧑⋈^{c_t}} S_2$ selects all nodes in S_1 that have descendants in S_2. More precisely, it inserts a node-cost tuple $(u_D, c, \tilde{c})$ into the result set if there is a tuple $(u_D, c_1, \tilde{c}_1) \in S_1$, and if there is a tuple $(v_D, c_2, \tilde{c}_2) \in S_2$ such that v_D is a descendant of u_D. If only one descendant of u_D exists in S_2, then the cost c is the sum of the cost c_1 assigned to the ancestor node, the cost c_2 assigned to the descendant node, the transformation cost c_t, and the *node distance* between u_D and v_D.

Definition 6.6 (Node distance) *The distance between two nodes u_D and v_D, denoted by nodedist(u_D, v_D), is the sum of the insertion costs of all nodes on the path between u_D and v_D (excluding u_D and v_D). If no path between u_D and v_D exists, then the distance is infinite.*

The node distance corresponds to the total cost of inserting nodes between two query-tree nodes in our theoretical model. However, the join operator does not insert the nodes explicitly, but rather checks how many insertions would be necessary in order to transform a part of a query tree so that it can be mapped to the path between the data-tree nodes u_D and v_D.

Often, u_D has more than one descendant in S_2. In this case, c is the lowest cost among the costs calculated for all descendants of u_D. In view of our theoretical model, this corresponds to the selection of the lowest-cost query subtree that can be embedded into the data subtree rooted at u_D, presumed that all costs in S_2 are minimal. We discuss the propagation of minimal costs through nested operators in Section 6.4.

Definition 6.7 (Join) *Let S_1 and S_2 be sets of node-cost tuples and c_t be a transformation cost. The join between the ancestor set S_1 and the descendant set S_2 is defined as*

$$S_1 \mathbin{⧑⋈^{c_t}} S_2 = \{\, (u_D, c_t + c_1 + c_{min}, \tilde{c}_1) \mid (u_D, c_1, \tilde{c}_1) \in S_1 \wedge (v_D, c_2, \tilde{c}_2) \in S_2 \wedge u_D \rightsquigarrow v_D \,\},$$

where $c_{min} = \min\{\, c_2 + \mathit{nodedist}(u_D, v_D) \mid (v_D, c_2, \tilde{c}_2) \in S_2 \wedge u_D \rightsquigarrow v_D \,\}$.

The result set of a join contains only the tuples from S_1 that have descendants in S_2. In contrast, the outerjoin operator $S_1 \mathbin{⟕^{c_t}_{c_d}} S_2$ inserts *all* tuples from S_1 into the result set,

independently of whether they have descendants in S_2 or not. The cost c_d passed to the outerjoin operator is the deletion cost defined for the nodes in S_2. If no descendant for a particular node u_D exists, or if c_d is lower than the embedding cost c_2 of v_D plus the node distance between u_D and v_D, then the operator adds c_d to the cost of the created tuple.

Definition 6.8 (Outerjoin) *Let S_1 and S_2 be sets of node-cost tuples, c_t be a transformation cost, and c_t be a deletion cost. The outerjoin between S_1 and S_2 is defined as*

$$S_1 \mathbin{⟕}_{c_d}^{c_t} S_2 = \{\, (u_D, c_t + c_1 + c_{min}, \tilde{c}_1) \mid (u_D, c_1, \tilde{c}_1) \in S_1 \,\},$$

where $c_{min} = \min(c_d, \min\{\, c_2 + \mathit{nodedist}(u_D, v_D) \mid (v_D, c_2, \tilde{c}_2) \in S_2 \wedge u_D \rightsquigarrow v_D \,\})$.

If S_2 stores the matches of a query selector for which an insertion restriction or relaxation is defined, then the cost calculation of the two join operators is modified. We describe the modifications informally: An insertion restriction, represented by the annotation insres of the s-node created for the selector, forbids the insertion of nodes between the nodes in S_1 and in S_2. The modified join and outerjoin operators discard all ancestor-descendant pairs u_D, v_D for which $\mathit{nodedist}(u_D, v_D) > 0$ holds. An insertion relaxation modifies the function *nodedist*. For each pair of ancestor-descendant nodes, the function returns a zero cost.

6.3.4 Union and Intersection

The union operator $S_1 \sqcup^{c_t} S_2$ creates the union of the node-cost tuples in the sets S_1 and S_2. If a node u_D appears in only one operand set, say $(u_D, c_1, \tilde{c}_1) \in S_1$, then the cost calculated for the resulting node-cost tuple is the sum of c_1 and the transformation cost c_t. If a node is in both S_1 and S_2, then the minimum of the operand cost is chosen and increased by c_t.

Definition 6.9 (Union) *Let S_1 and S_2 be sets of node-cost tuples, and c_t be a transformation cost. The union between S_1 and S_2 is defined as*

$$S_1 \sqcup^{c_t} S_2 = \{\, (u_D, c_t + c_1, \tilde{c}_1) \mid (u_D, c_1, \tilde{c}_1) \in S_1 \cup S_2 \wedge \nexists (u_D, c_2, \tilde{c}_2) \in S_1 \cup S_2 : c_2 < c_1 \,\}.$$

Note that the backup costs of corresponding operands are equal if the union operator is used in query-execution plans (the backup costs represent the same primary cost propagated along different paths). Therefore, if both operands exist, then the backup cost of the first one is used.

The intersection operator $S_1 \sqcap^{c_t} S_2$ creates a set of all tuples $(u_D, c, \tilde{c})$ such that u_D appears in both operand sets. The cost c is the sum of the costs assigned to the corresponding nodes plus c_t.

Definition 6.10 (Intersection) *Let S_1 and S_2 be sets of node-cost tuples and c_t be a transformation cost. The intersection between S_1 and S_2 is defined as*

$$S_1 \sqcap^{c_t} S_2 = \{\, (u_D, c_t + c_1 + c_2 - \tilde{c}_1, \tilde{c}_1) \mid (u_D, c_1, \tilde{c}_1) \in S_1 \wedge (u_D, c_2, \tilde{c}_2) \in S_2 \,\}.$$

Again, the backup costs $\tilde{c}_1, \tilde{c}_2$ of the corresponding operands are required to be equal. They are initialized by selections, and store the summand common to the primary costs c_1, c_2. The subtraction of $\tilde{c}_1$ filters out this common summand.

6.3.5 Operator Equivalences

In this subsection, we investigate equivalences between operators. These equivalences are useful for query optimization; they are also important to show that a query can be evaluated without the creation of its disjunctive normal form. The operator equivalences are listed in the following lemma.

Lemma 6.1 (Operator equivalences) *The following equivalences for operators hold:*

$$S_1 \sqcap^{c} S_2 = S_2 \sqcap^{c} S_1 \tag{6.1}$$

$$S_1 \sqcup^{c} S_2 = S_2 \sqcup^{c} S_1 \tag{6.2}$$

$$S_1 \sqcap^{c_a} (S_2 \sqcap^{c_b} S_3) = (S_1 \sqcap^{c_a} S_2) \sqcap^{c_b} S_3 \tag{6.3}$$

$$S_1 \sqcup^{c_a} (S_2 \sqcup^{c_b} S_3) = (S_1 \sqcup^{c_a} S_2) \sqcup^{c_b} S_3 \tag{6.4}$$

$$S_1 \sqcap^{c_a} (S_2 \sqcup^{c_b} S_3) = (S_1 \sqcap^{0} S_2) \sqcup^{c_c} (S_1 \sqcap^{0} S_3), \text{ where } c_a + c_b = c_c \tag{6.5}$$

$$S_1 \ltimes\!\!\!\rhd^{c_a} (S_2 \sqcup^{c_b} S_3) = (S_1 \ltimes\!\!\!\rhd^{0} S_2) \sqcup^{c_c} (S_1 \ltimes\!\!\!\rhd^{0} S_3), \text{ where } c_a + c_b = c_c \tag{6.6}$$

$$(S_1 \ltimes\!\!\!\rhd^{c_a} S_2) \ltimes\!\!\!\rhd^{c_b} S_3 = (S_1 \ltimes\!\!\!\rhd^{c_c} S_3) \ltimes\!\!\!\rhd^{c_d} S_2, \text{ where } c_a + c_b = c_c + c_d \tag{6.7}$$

If $\forall (u_D, c_1, \tilde{c}_1) \in S_1 : c_1 = \tilde{c}_1$, then the following equivalences hold in addition:

$$S_1 \sqcup^{c_a} (S_2 \sqcap^{0} S_3) = (S_1 \sqcup^{c_b} S_2) \sqcap^{c_c} (S_1 \sqcup^{c_d} S_3), \text{ where } c_a = c_b + c_c + c_d \tag{6.8}$$

$$(S_1 \ltimes\!\!\!\rhd^{c_a} S_2) \ltimes\!\!\!\rhd^{c_b} S_3 = (S_1 \ltimes\!\!\!\rhd^{c_c} S_2) \sqcap^{c_d} (S_1 \ltimes\!\!\!\rhd^{c_e} S_3), \text{ where } c_a + c_b = c_c + c_d + c_e \tag{6.9}$$

If further nodes$(S_1) \supseteq$ nodes(S_3), then the following equivalence holds in addition:

$$(S_1 \ltimes\!\!\!\rhd^{c_a} S_2) \sqcap^{c_b} S_3 = S_3 \ltimes\!\!\!\rhd^{c_c} S_2, \text{ where } c_a + c_b = c_c \tag{6.10}$$

Proof: Equivalences 6.1–6.4 hold because "+" and "min" are commutative and associative.

Equivalence 6.5: Both sides select the same set of nodes because

$$nodes(S_1) \cap (nodes(S_2) \cup nodes(S_3)) = (nodes(S_1) \cap nodes(S_2)) \cup (nodes(S_1) \cap nodes(S_3))$$

holds. The intersection operator adds the costs of corresponding operands; the union selects the minimum. For cost calculation, three cases are possible: First, a node appears in S_1 with cost c_1 and in S_2 with cost c_2, but not in S_3. On the left side of the equation, the union passes $(u_D, c_2 + c_b, \tilde{c}_2)$ to the intersection operator, which calculates $c_1 + c_a + c_2 + c_b - \tilde{c}_1$. On the right equation side, the right intersection fails, and the left intersection calculates $c_1 + c_2 - \tilde{c}_1$. The union adds c_c to this sum. Because $c_a + c_b = c_c$, it holds $c_1 + c_a + c_2 + c_b - \tilde{c}_1 = c_1 + c_2 - \tilde{c}_1 + c_c$. Second, the roles of S_2 and S_3 are reversed, and the same argumentation holds. Third, $(u_D, c_3, \tilde{c}_3) \in S_3$ additionally holds. The left side calculates $c_1 + c_a + \min(c_2, c_3) + c_b - \tilde{c}_1$, and the right side calculates $\min(c_1 + c_2 - \tilde{c}_1, c_1 + c_3 - \tilde{c}_1) + c_c$. The equivalence holds because $x + \min(y, z) = min(x + y, x + z)$ holds for all values of x, y, z.

Equivalence 6.6: Consider the left side of the equation. A node u_D in S_1 is inserted into the result set if and only if it has descendants in either S_2 or S_3. The same holds at the right side: u_D is in the result set of $S_1 \mathbin{|\!\bowtie\!|} S_2$ if and only if it has a descendant in S_2. It is in the result set of $S_1 \mathbin{|\!\bowtie\!|} S_3$ if and only if it has a descendant in S_3. It is in the final result set if and only if it is in the result sets of $S_1 \mathbin{|\!\bowtie\!|} S_2$ or $S_1 \mathbin{|\!\bowtie\!|} S_3$. For cost calculation, four cases are possible: First, u_D has descendants in S_2, but not in S_3. Let c_2 be the cost of the lowest-cost descendant v_D of u_D in S_2. The left side calculates $c_a + c_1 + (c_b + c_2) + nodedist(u_D, v_D)$; the right side calculates $c_c + (0 + c_1 + c_2 + nodedist(u_D, v_D))$. Because $c_a + c_b = c_c$ holds, both sides are equivalent. Second, the roles of S_2 and S_3 are reversed, and the same argumentation holds. Third, u_D has descendants in both S_2 and S_3, and the same node is the lowest-cost descendant of u_D in both sets. Let v_D be this node and let c_2 and c_3 be its costs in S_2 and S_3, respectively. The left side calculates $c_a + c_1 + (c_b + \min(c_2, c_3)) + nodedist(u_D, v_D)$; the right side calculates $c_c + \min(0 + c_1 + c_2 + nodedist(u_D, v_D), 0 + c_1 + c_3 + nodedist(u_D, v_D))$. Both sides are equivalent. Fourth, u_D has descendants in both S_2 and S_3, but the lowest-cost descendants of u_D are different nodes. Let v_D, w_D be the lowest-cost descendants of u_D in S_2, S_3 with costs c_2, c_3. The left side calculates $c_a + c_1 + \min((c_b + c_2) + nodedist(u_D, v_D), (c_b + c_3) + nodedist(u_D, w_D))$; and the right side calculates $c_c + \min(0 + c_1 + c_2 + nodedist(u_D, v_D), 0 + c_1 + c_3 + nodedist(u_D, w_D))$. Both sides are equivalent.

Equivalence 6.7: The innermost join on the left equation side inserts a node-cost tuple $(u_D, c_1, \tilde{c}_1) \in S_1$ into the result set if and only if u_D has a descendant in S_2. The outermost join inserts u_D into the final result set if and only if it also has descendants in S_3. A

sequence of such descendant tests can be ordered arbitrarily. Let v_D be the lowest-cost descendant of u_D in S_2 with cost c_2, and let w_D be the lowest-cost descendant of u_D in S_3 with cost c_3. The left side calculates $(c_a + c_1 + c_2 + \mathit{nodedist}(u_D, v_D)) + c_b + c_3 + \mathit{nodedist}(u_D, w_D)$, and the right side calculates $(c_c + c_1 + c_3 + \mathit{nodedist}(u_D, w_D)) + c_d + c_2 + \mathit{nodedist}(u_D, v_D)$. Because $c_a + c_b = c_c + c_d$ holds by definition, both sides are equivalent.

Equivalence 6.8: Both sides select the same set of nodes because

$$\mathit{nodes}(S_1) \cup (\mathit{nodes}(S_2) \cap \mathit{nodes}(S_3)) = (\mathit{nodes}(S_1) \cup \mathit{nodes}(S_2)) \cap (\mathit{nodes}(S_1) \cup \mathit{nodes}(S_3))$$

holds. For cost calculation, four cases are possible: First, a node appears in S_1 with cost c_1, but not in S_2 and S_3. The left side calculates $c_1 + c_a$ and the right side calculates $c_1 + c_b + c_c + c_d$. Both sides are equivalent by definition. Second, a node is in S_1 with cost c_1 and in S_2 with cost c_2, but not in S_3. On the left side the intersection fails and the resulting cost is $c_1 + c_a$. On the right side, the unions calculate $c_1 + c_b$ and $c_1 + c_d$ as primary cost and $\tilde{c}_1$ as backup cost, respectively. The following intersection calculates $c_1 + c_b + c_1 + c_d - \tilde{c}_1 + c_c$. Because by assumption $c_1 = \tilde{c}_1$ and $c_a = c_b + c_c + c_d$, both sides are equal. Third, the roles of S_2 and S_3 are reversed, and the same argumentation as for case two holds. Fourth, $(u_D, c_3, \tilde{c}_3) \in S_3$ additionally holds. The left side calculates $\min(c_1, c_2 + c_3 - \tilde{c}_2) + c_a$. The backup cost $\tilde{c}_2$ is by assumption equal to $\tilde{c}_1$, and is therefore, according to the precondition, equal to c_1. Because both unions on the right side have the backup cost $\tilde{c}_1$, the result of the following intersection is $\min(c_1, c_2) + c_b + \min(c_1, c_3) + c_d - \tilde{c}_1 + c_c$. It holds

$$\begin{aligned}
\min(c_1, c_2 + c_3 - \tilde{c}_2) + c_a &= \min(c_1, c_2) + c_b + \min(c_1, c_3) + c_d - \tilde{c}_1 + c_c \\
\min(c_1, c_2 + c_3 - c_1) + c_a &= \min(c_1, c_2) + \min(c_1, c_3) - c_1 + c_a \\
\min(2 \cdot c_1, c_2 + c_3) + c_a &= \min(c_1, c_2) + \min(c_1, c_3) + c_a
\end{aligned}$$

for all values of c_1, c_2, c_3.

Equivalence 6.9: Consider the left side of the equation. A node u_D in S_1 is inserted into the result set if and only if it has descendants in both S_2 and S_3. The same holds at the right side: u_D is in the result set of $S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2$ if and only if it has a descendant in S_2. It is in the result set of $S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_3$ if and only if it has a descendant in S_3. It is in the final result set if and only if it is in both the result sets of $S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2$ and $S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_3$. Let c_2 (c_3) be the cost of the lowest-cost descendant of u_D in S_2 (S_3). The left side calculates $(c_1 + c_2 + c_a) + c_3 + c_b$, and the right side calculates $(c_1 + c_2 + c_c) + (c_1 + c_3 + c_d) - \tilde{c}_1 + c$. Because $c_1 = \tilde{c}_1$, and $c_a + c_b = c_c + c_d + c$, both sides are equivalent.

Equivalence 6.10: It holds $\mathit{nodes}(S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2) \supseteq \mathit{nodes}(S_3 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2)$ because we assume $\mathit{nodes}(S_1) \supseteq \mathit{nodes}(S_3)$. All nodes that are in $\mathit{nodes}(S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2)$ but not in $\mathit{nodes}(S_3 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2)$ are discarded during the following intersection with S_3. It follows that $\mathit{nodes}((S_1 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2) \sqcap S_3) = \mathit{nodes}(S_3 \mathbin{\text{I}\!\bowtie\!\triangleleft} S_2)$.

Let u_D be a node represented by tuples $(u_D, c_1, \tilde{c}_1) \in S_1$ and $(u_D, c_3, \tilde{c}_3) \in S_3$ (therefore, u_D is also in the result set). Let v_D be the lowest-cost descendant of u_D in S_2 with cost c_2. The left equation side calculates $(c_1 + c_2 + \mathit{nodedist}(u_D, v_D) + c_a) + c_3 - \tilde{c}_3 + c_b$. The right side calculates $c_3 + c_2 + \mathit{nodedist}(u_D, v_D) + c_c$. By assumption, $c_1 = \tilde{c}_1$ and $c_a + c_b = c_c$. Because we also demand that the backup costs of corresponding nodes are equal, it holds $\tilde{c}_1 = \tilde{c}_3$. It follows that both sides calculate the same cost. □

All equivalences also hold if some or all join operators are replaced by outerjoin operators. We omit the proofs for these cases. The precondition of Equivalences 6.9 and 6.8 seems to be a restriction of their usability. However, this precondition always holds in the query-execution plans that we introduce in the following section. Based on Equivalences 6.5 and 6.9, we show in Section 6.5.2 that Boolean queries can be evaluated "in place" without the creation of the disjunctive normal form. Moreover, even the preconditions of Equivalence 6.10 are fulfilled in most cases. Therefore, this rule is a powerful means for reducing the number of operators in query-execution plans.

6.4 Query-Execution Plans

In Section 6.2, we proposed the expanded representation of a query as a construction directive for the closure of a query. In this section, we show how expanded representations can be used to construct *query-execution plans*. The task of a query-execution plan is to assemble an algorithm that computes, for each logical document (subtree) of the data tree, the embedding cost of the lowest-cost query tree in the closure — without actually creating the closure.

Definition 6.11 (Query-execution plan) *A query-execution plan is a binary rooted DAG* $\mathcal{P} = (N, E, r, op, param, delcost, transcost)$*, where* N *is a set of nodes,* $E = (N \times N)$ *is a set of edges,* r *is the root of* $\mathcal{P}$*, and*

$$op : N \to \{\sigma, \ltimes, \rtimes, \sqcup, \sqcap\}, \qquad delcost : N \to \mathbb{R}^+,$$
$$param : N \to \mathcal{T} \times P^* \times D^*, \qquad transcost : N \to \mathbb{R}^+$$

are functions, where $\sigma, \ltimes, \rtimes, \sqcup, \sqcap$ *are the operators defined in Section 6.3,* $\mathcal{T}$ *is a finite set of types,* $D^* = \bigcup_{(D,P)\in\mathcal{T}} D$ *is the union of domains in* $\mathcal{T}$*,* $P^* = \bigcup_{(D,P)\in\mathcal{T}} P$ *is the union of predicates in* $\mathcal{T}$*, and* $\mathbb{R}^+$ *is the set of non-negative real numbers. For all* $u_\mathcal{P} \in N$ *holds:* $op(u_\mathcal{P}) = \sigma$ *if and only if* $u_\mathcal{P}$ *is a leaf. The function param is defined only for leaves.*

A query execution plan is a *binary* DAG because all inner nodes have two children (the operands of the operator assigned to a node). We write $\mathit{child}_1(u_\mathcal{P})$ and $\mathit{child}_2(u_\mathcal{P})$ to refer

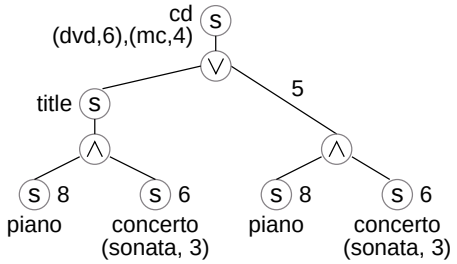


Figure 6.8: The expanded query representation for `cd/title["piano" and "concerto"]`, where the following transformations are allowed: deletions of title (cost 5), piano (cost 8), and concerto (cost 6); value changes of cd to dvd (cost 6) or mc (cost 4), and concerto to sonata (cost 3); arbitrary insertions.

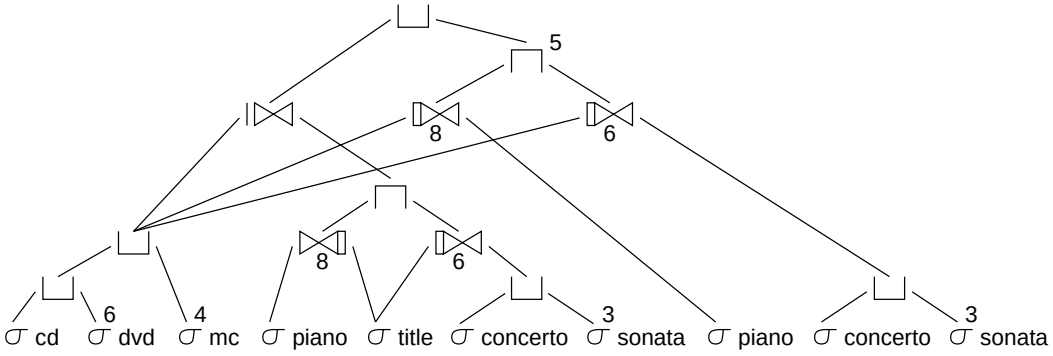


Figure 6.9: The query-execution plan for the expanded representation shown in Figure 6.8.

to the children of a plan node $u_{\mathcal{P}}$. A subgraph of a plan consisting of all nodes and edges reachable from a certain node is called a *subplan*. The functions *delcost* and *transcost* are counterparts to the functions of an expanded representation (see Definition 6.1 on page 69): The function *delcost* returns the deletion costs of query selectors. For example, if for a node $u_{\mathcal{P}}$ holds $\mathit{delcost}(u_{\mathcal{P}}) = 11$ and $\mathit{op}(u_{\mathcal{P}}) = ⟕$, then the cost 11 is passed as parameter c_d to the outerjoin operator. Otherwise, if $\mathit{op}(u_{\mathcal{P}}) \neq ⟕$, then the cost is ignored. The function *transcost* assigns transformation costs to the operators. A transformation cost may be the deletion cost of an inner s-node, a permutation cost, or value-change cost. The function *param* defines the parameters for the selection operators. For example, if $\mathit{transcost}(u_{\mathcal{P}}, v_{\mathcal{P}}) = 4$, $\mathit{op}(v_{\mathcal{P}}) = \sigma$, and $\mathit{param}(v_{\mathcal{P}}) = (\mathit{integer}, \leq, 2000)$, then $u_{\mathcal{P}}$ represents the selection operator $\sigma^4[\mathit{integer}, \leq, 2000]$.

Figure 6.9 shows an example of a query-execution plan. This plan is constructed for the expanded query representation depicted in 6.8. The simple example covers all main components of query-execution plans: Each s-node in the expanded representation is modeled as a selection operator, which finds all data-tree nodes matching the query selector represented by the

s-node. Depending on the number of paths leading to an s-node, several identical selection operators may exist (two for piano and concerto). If an s-node has alternative values (like cd or concerto), then a separate selection operator exists for each of these values. All selection operators for the same s-node are connected by union operators. The edges leading to the dvd and mc selectors are annotated with the value-change costs. For each parent-child relationship of two s-nodes in the expanded representation (ignoring interspersed $\wedge$ and $\vee$-nodes), a join ensures that the corresponding data-tree nodes have an ancestor-descendant relationship. This implicit relaxation corresponds to a virtual insertion of nodes. Because a join operator is able to measure the node distance between an ancestor and a descendant, we know how large the total insertion cost of all nodes on the path between the ancestor and the descendant is. Leaf deletions are implemented by outerjoin operators. For example, the lower-left outerjoin passes a title node to the result set even if it does not have descendants with the value piano. In this case, the cost 8 is added to the primary cost of the tuple. Intersections between the joins ensure that only those data-tree nodes remain in the result set of an s-node that have a descendant with respect to each s-child of this s-node. The top-level union operator models the choice between keeping the inner node title or deleting it. If no title node for a particular document exists (or its deletion is cheaper than its preservation), then the tuple computed by the right subplan is passed to the result set, and its primary cost is incremented by 5. Because permutations are modeled like deletions in an expanded representation, they are also modeled like deletions in an execution plan.

Algorithm 6.4 on the facing page implements the mapping of an expanded query representation to a query-execution plan. We use a simplified notation to highlight the way a plan is constructed. For example, the expression $\mathcal{P} := \sigma[type(v_E), pred(v_E), value(v_E)]$ is a abbreviation for adding a new node $v_\mathcal{P}$ to $\mathcal{P}$ and defining $op(v_E) := \sigma$ and $param(v_\mathcal{P}) := (type(v_E), pred(v_E), value(v_E))$. The algorithm traverses the nodes in the expanded query representation top-down. It creates the selection parts of the plan during its descent, and then combines them with join, intersection, and union operators during its ascent. If the currently processed node v_E is an s-node, then a selection operator is created, and type, predicate, and value of the selector represented by v_E are passed to the operator (Line 2). If v_E is annotated with value-cost pairs, then an additional selection operator is created for each pair and initialized with the alternative value (Line 4). The value-change cost is passed as the transformation cost to the operator, i.e., the function *transcost* of the node the operator belongs to is extended. Union operators connect each new selection operator with the plan constructed in the previous step. If v_E has a child (which may be an s-node, a $\wedge$-node, or a $\vee$-node), then the function is recursively called passing the child node and the plan constructed so far (Line 6). If v_E is the child of another s-node, then a plan $\mathcal{P}_A$ representing

Algorithm 6.4 creates an execution plan based on the expanded representation of a query.

function `create_plan`$(v_E, \mathcal{P}_A)$

input: v_E – a node in an expanded query representation,
$\mathcal{P}_A$ – a plan for the selection of the ancestor set.
returns: the plan created for the subtree rooted at v_E.

```
1:  if op(v_E) = s then
2:     Create a plan P := σ[type(v_E), pred(v_E), value(v_E)]
3:     foreach (α, c) ∈ values(v_E) do
4:        Create a plan P := P ⊔ σ^c[type(v_E), pred(v_E), α]
5:     if v_E has a child then
6:        P := create_plan(child(v_E), P)
7:     if v_E is not a top-level s-node then
8:        if v_E is a leaf and delcost(v_E) < ∞ then
9:           Create a plan P := P_A ⟕_{c_d} P, where c_d = delcost(v_E)
10:       else
11:          Create a plan P := P_A ⋈ P
12:       Adjust the cost calculation of the join according to mod(v_E)
13: else /* v_E is a ∧-node or a ∨-node */
14:    P_L := create_plan(child_1(v_E), P_A)
15:    P_R := create_plan(child_2(v_E), P_A)
16:    case op(v_E) of
17:       ∧: Create a plan P := P_L ⊓ P_R
18:       ∨: Create a plan P := P_L ⊔ P_R^{c_t}, where c_t = transcost(v_E, child_2(v_E))
19: return P
```

the selection of the ancestor nodes already exists. This plan is connected with the plan for the descendant nodes using a join operator (Lines 8–11). The cost function of the new join operator is adjusted with the help of the attribute $mod(v_E)$, which may indicate an insertion restriction or relaxation (Line 12). The choice of the appropriate join operator depends on whether v_E is a leaf that is allowed to be deleted (outerjoin) or an inner node (normal join). If v_E is either a $\wedge$-node or a $\vee$-node, then the subplans for the children of v_E are created first (Lines 14 and 15). Again, the plan $\mathcal{P}_A$ is passed so that it can be used as an ancestor plan if the next s-node is reached. The subplans returned by the recursive calls are connected by an intersection or union operator (Lines 17 and 18). In the latter case, an edge cost is assigned to the node, which defines the transformation cost to be passed to the top-level operator of the right subplan. We use the notation $\mathcal{P}_R^{c_t}$ to indicate that the top-level node in $\mathcal{P}_R$ is annotated with the transformation cost c_t.

Let Q be a query, T_E be its expanded representation, r be the root of T_E, and ε be an empty plan. Then the call

$$\mathcal{P} := \texttt{create\_plan}(r, \varepsilon)$$

returns an execution plan $\mathcal{P}$ for Q. For convenience, we define the abbreviation

$$\mathcal{P} := \texttt{create}(Q)$$

for the creation of an expanded query representation and its mapping to an execution plan.

Algorithm 6.5 shows an evaluation algorithm for query execution plans. It traverses a plan top-down until a selection operator (assigned to a leaf) is reached, passes the parameters to the operator, and returns the result (Line 3). For each other operator (assigned to an inner node), the algorithm executes its operands (assigned to the children of the node) before the operator itself is executed (Lines 4–9). The outerjoin operator needs special treatment because it takes the additional parameter c_d (Lines 6–8). In Section 6.6.2, we present an optimized version of this algorithm.

Algorithm 6.5 evaluates a query-execution plan.

---

function $\texttt{evaluate}(\mathcal{P}, v_\mathcal{P}, T_D)$

params: $\mathcal{P} = (N, E, r, op, param, delcost, transcost)$ – a query-execution plan,
$v_\mathcal{P}$ – a node in N,
T_D – a data tree,

returns: the results of the subplan of $\mathcal{P}$ with root $v_\mathcal{P}$.

1: $c_t := transcost(v_\mathcal{P})$
2: **if** $op(v_\mathcal{P}) = \sigma$ **then**
3: **return** $\sigma^{c_t}[param(v_\mathcal{P})]\ T_D$
4: $S_1 := \texttt{evaluate}(\mathcal{P}, child_1(v_\mathcal{P}), T_D)$
5: $S_2 := \texttt{evaluate}(\mathcal{P}, child_2(v_\mathcal{P}), T_D)$
6: **if** $op(v_\mathcal{P}) = ⟕$ **then**
7: $c_d := delcost(v_\mathcal{P})$
8: **return** $S_1\ ⟕_{c_d}^{c_t}\ S_2$
9: **return** $S_1\ op(v_\mathcal{P})^{c_t}\ S_2$

Let Q be a query, $\mathcal{P}$ be an execution plan for Q, r be the root of $\mathcal{P}$, and T_D be a data tree. Then the algorithm

$$S := \texttt{evaluate}(\mathcal{P}, r, T_D)$$

finds the set S of node-cost tuples representing the results of the query for which $\mathcal{P}$ is constructed. We additionally assume a function $\mathsf{sort}(S, n)$ that sorts the node-cost tuples in S by increasing primary costs, and selects the n tuples with the lowest costs. Then

$$S := \texttt{sort}(\texttt{evaluate}(\mathcal{P}, r, T_D), n)$$

returns the set S of the best n node-cost tuples. In the following, we omit the parameter r needed for recursive calls and define

$$\texttt{evaluate}(\mathcal{P}, T_D) \stackrel{\text{def}}{=} \texttt{evaluate}(\mathcal{P}, r, T_D).$$

6.5 The Equivalence of Theoretical and Practical Query Evaluation

The all-results problem (see Definition 5.12 on page 62) defines the set of results to be returned for a given query Q and a given data tree T_D. In this section, we prove that the algorithm `evaluate`$(\mathcal{P}, T_D)$ solves the all-results problem, where $\mathcal{P}$ is the query-execution plan constructed for Q. A simple consequence is that the algorithm `sort`(`evaluate`$(\mathcal{P}, T_D), n)$ solves the best-n-results problem (Definition 5.13 on page 62). To keep the proofs simple, we do not consider query restrictions and relaxations. The proofs for these modifications of the default semantics are analogous.

6.5.1 Conjunctive Queries

We first consider execution plans for conjunctive queries, and assume that insertions are allowed, but not deletions, permutations, or value changes. To prove that the evaluation of a plan created for a conjunctive query solves the all-results problem, we emphasize the main principles underlying the evaluation of a plan, and relate these principles to the theoretical model defined in Chapter 5.

Let Q be an arbitrary conjunctive query, and T_D be an arbitrary data tree. In the theoretical model, we create the separated representation $\mathbb{Q}$ of Q. Because Q is conjunctive, $\mathbb{Q}$ consists of a single query tree. From $\mathbb{Q}$, we derive the closure $\mathbb{Q}^*$ of transformed query trees, and explicitly embed each query tree $T_Q \in \mathbb{Q}^*$ into T_D. In practice, we create the expanded representation T_E of Q, which resembles the single query tree in $\mathbb{Q}$, but uses $\wedge$-nodes to model conjunctions. We find the embeddings of transformed query trees by selecting all matches[1] of the s-nodes in T_E such that parent-child relationships in T_E are mapped ancestor-descendant relationships in T_D. Figure 6.10 on the following page illustrates the relationships between the theoretical model (left) and the practical approach.

The following definition formalizes the matching model used for the construction and evaluation of query-execution plans. Based on this definition, we prove in Lemma 6.2 on the next page that the theoretical query-evaluation method and its practical counterpart yield the same set of embedding images.

Definition 6.12 (Selection of embedding images) *Let Q be a query, T_E be the expanded representation of Q, and T_D be a data tree. Let S_E be the set of trees included in T_D where*

[1] Recall that a match of a query-tree node u_Q (or the s-node in T_E representing u_Q) is a data-tree node that has the same type as u_Q and fulfills the selection predicate of u_Q.

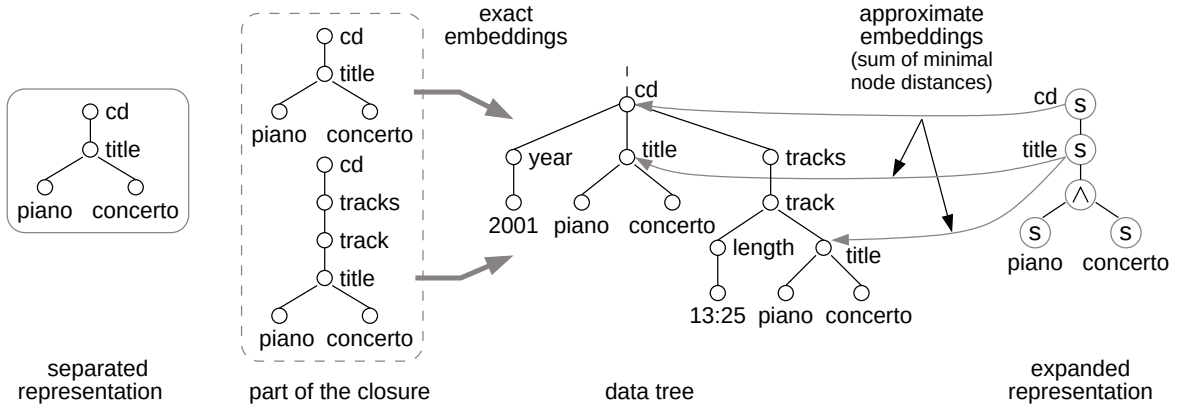


Figure 6.10: The relationships between the theory and practice of evaluating conjunctive queries for which only insertions are allowed. Example query: `cd/title["piano and "concerto"]`. In the theoretical model, the separated representation is created, and the closure of transformed query trees is derived (partially shown in the figure). For each query tree, the exact embeddings are selected. In the practical approach, the expanded representation is constructed, and the approximate embeddings are selected by expanding the edges leading to s-nodes.

for each $T'_D \in S_E$ holds: (i) T'_D contains a match of each s-node in T_E, (ii) if two s-nodes in T_E have a parent-child relationship, then their matches in T'_D are connected by a path, and (iii) no other nodes are in T'_D.

We call each tree included in T_D selected by Q according to Definition 6.12 the *image* of an *approximate embedding* of Q (or T_E, equivalently) into T_D.

Lemma 6.2 (Selection equivalence) *Let Q be a query, $\mathbb{Q}^*$ be the closure of Q, T_E be the expanded representation of Q, and T_D be a data tree. Let S_E be the set constructed for T_E and T_D according to Definition 6.12. Let $S_{\mathbb{Q}^*}$ be the set of trees included in T_D that are embedding images of query trees in $\mathbb{Q}^*$. Then $S_E = S_{\mathbb{Q}^*}$ holds.*

Proof: Let $\mathbb{Q} = \{ T_Q \}$ be the separated representation of Q. Recall that T_Q and T_E have analogous structures, except that in T_Q conjunctions are implicit, whereas in T_E additional $\wedge$-nodes exist.

Case $S_E \subseteq S_{\mathbb{Q}^*}$: Let $T'_D \in S_E$ be an arbitrary tree. We show that there is a transformed query tree $T'_Q \in \mathbb{Q}^*$ with the embedding image T'_D. If T'_Q exists, then T'_D is by definition in $S_{\mathbb{Q}^*}$. According to Definition 6.12, T'_D is selected because it consists of matches of all s-nodes in T_E (or nodes in T_Q, respectively). If two s-nodes in T_E (nodes in T_Q) have a parent-child

relationship, then their matches in T'_D have an ancestor-descendant relationship. We can derive T'_Q from T_Q by inserting nodes between each pair of parent-child nodes u_Q, v_Q in a way that the created path matches the path between u_D and v_D in T'_D. Because embeddings need not to be injective (see Definition 5.1 on page 53), images of query paths may overlap arbitrarily, and therefore T'_D is an embedding image of T'_Q. Because we can construct T'_Q by inserting nodes into T_Q, T'_Q is in $\mathbb{Q}^*$. Therefore, $T'_D \in S_{\mathbb{Q}^*}$ holds.

Case $S_E \supseteq S_{\mathbb{Q}^*}$: Let $T'_D \in S_{\mathbb{Q}^*}$ be an arbitrary tree. We show that T'_D is in S_E. It exists a query tree $T'_Q \in \mathbb{Q}^*$ of which T'_D is the embedding image. T'_Q has the following structure: For each node in T_Q, there is a node with the same properties in T'_Q (called *core node* in the following). If two nodes u_Q and v_Q in T_Q have a parent-child relationship, then their corresponding nodes u'_Q and v'_Q in T'_Q have an ancestor-descendant relationship. There exist either a single edge between u'_Q and v'_Q or a path of inserted nodes. Paths belonging to different descendants of u'_Q do not share common nodes because insertions are restricted (see Definition 5.5 on page 57). Each core node in T'_Q must match exactly one node in T'_D. A path between two core nodes in T'_Q must map to a path between the matches of the core nodes. No further restrictions exist. In particular, paths of the embedding image may overlap because Definition 5.1 does not require the embedding function to be injective. According to Definition 6.12, only trees that have exactly the properties described above are inserted into S_E. Therefore, $T'_D \in S_E$ holds. □

We now investigate the selection of the lowest-cost embedding with respect to a logical document $T_D[u_D]$ of a data tree T_D. Let Q be a conjunctive query, $\mathbb{Q}^*$ be its closure, and T_E be its expanded representation. In the theoretical model, we select all query trees in $\mathbb{Q}^*$ that can be embedded into T_D such that they return $T_D[u_D]$ as the result. Among these query trees, we select the one with the lowest embedding cost. In practice, we select the embedding image (according to Definition 6.12) that has the lowest cost among all embedding images with root u_D. The cost of an embedding image is the sum of the node distances between the matches of the s-nodes in T_E. We formally define the cost of an embedding image:

Definition 6.13 (Cost of an embedding image) *Let Q be a query, T_E be the expanded representation of Q, T_D be a data tree, and S_E be the set of embedding images according to Definition 6.2. Let $T'_D \in S_E$ be an embedding image, and N_E be the set of nodes in T'_D that are matches of s-nodes in S_E. The cost of T'_D is defined as*

$$imgcost(T'_D) = \sum_{(u_D, v_D) \in (N_E \times N_E)} nodedist(u_D, v_D).$$

Recall from Definition 6.6 on page 79 that the distance between two nodes not connected by a path is infinite. The following lemma shows that the cost calculation according to Definition 6.13 is correct:

Lemma 6.3 (Cost equivalence) *Let Q be a query, $\mathbb{Q}^*$ be the closure of Q, T_E be the expanded representation of Q, T_D be a data tree, and S_E be the set of embedding images according to Definition 6.2. Let $T'_Q \in \mathbb{Q}^*$ be a transformed query tree, and $T'_D \in S_E$ be its embedding image. Then $\mathit{embcost}(T'_Q) = \mathit{imgcost}(T'_D)$ holds.*

Proof: Let $\mathbb{Q} = \{ T_Q \}$ be the separated representation of Q. If $T'_Q = T_Q$, then the parent-child relationships in T'_Q are mapped to parent-child relationships in T'_D. Then, both costs are zero. Otherwise, T'_Q is derived from T_Q by replacing edges with nodes. The paths created by node insertions into T'_Q are mapped to paths in T'_D. Let u_Q and v_Q be nodes in T'_Q between which a sequence of nodes has been inserted. Let u_D and v_D be their matches. Because the corresponding nodes on the paths $u_Q \rightsquigarrow v_Q$ and $u_D \rightsquigarrow v_D$ have the same insertion costs, $\mathit{nodedist}(u_D, v_D)$ returns the total cost of the nodes inserted between u_Q and v_Q. Because the insertion costs of the nodes on all paths are equal, the sums are also equal. □

The following lemma uses the equivalences between the theoretical and the practical model to prove the completeness and soundness of the creation and evaluation of a plan.

Lemma 6.4 (Completeness/soundness of Algorithm 6.5 for conjunctive queries) *Let Q be a conjunctive query for which only insertions are allowed, T_E be the expanded representation of Q, $\mathcal{P}$ be the execution plan created for T_E, and T_D be a data tree. Then algorithm* `evaluate`$(\mathcal{P}, T_D)$ *solves the all-results problem for Q and T_D.*

Proof: We prove the lemma by induction over the subtrees of T_E whose roots are s-nodes (called *s-subtrees* in the following) in the order they are visited by Algorithm 6.4 on page 87. This order determines the structure of the plan and the order of its evaluation. For each s-node u_E in T_E, we show that the set S computed by the evaluation algorithm contains the roots of all subtrees of T_D that directly include[2] images of approximative embeddings of $T_E[u_E]$ into T_D (completeness). For each $(u_D, c, \tilde{c}) \in S$, we show that $T_D[u_D]$ directly includes images of approximate embeddings of $T_E[u_E]$ into T_D, and that c is the lowest cost of all these images (soundness). Note that for conjunctive queries, all operator superscripts are zero. Thus, we omit them in the proof.

[2]Recall from Section 4.1 that a tree includes another tree *directly* if both trees have the same root.

Base step: Node u_E is an s-leaf. The subplan created for $T_E[u_E]$ has the form $\bar{\mathcal{P}} = \sigma[\tau, \phi, \alpha]$, where τ, ϕ, α are the type, value, and selection predicate of u_E. Completeness: Algorithm `evaluate`$(\bar{\mathcal{P}}, T_D)$ returns a set S that consists of all nodes in T_D that are matches of u_E. Thus, the algorithm fulfills the requirements of Definition 6.12, and is complete according to Lemma 6.2. Soundness: Only matches of u_E (roots of embedding images) are in S. All costs in S are minimal (zero) because for each subtree of T_D at most one directly included image of an approximate embedding of $T_E[u_E]$ exists.

Hypothesis: $T_E[u_E]$ has the s-subtrees $T_E[v_{E_1}], T_E[v_{E_2}], \ldots, T_E[v_{E_m}]$ for which the subplans $\bar{\mathcal{P}}_1, \bar{\mathcal{P}}_2, \ldots, \bar{\mathcal{P}}_m$ exist. We assume that the algorithm $S_i :=$ `evaluate`$(\bar{\mathcal{P}}_i, T_D)$ is complete and sound $(1 \leq i \leq m)$.

Induction step: The subplan created for $T_E[u_E]$ has the form

$$\bar{\mathcal{P}} = (\bar{\mathcal{P}}_0 \mathbin{|\!\bowtie\!\triangleleft} \bar{\mathcal{P}}_1) \sqcap (\bar{\mathcal{P}}_0 \mathbin{|\!\bowtie\!\triangleleft} \bar{\mathcal{P}}_2) \sqcap \ldots \sqcap (\bar{\mathcal{P}}_0 \mathbin{|\!\bowtie\!\triangleleft} \bar{\mathcal{P}}_m),$$

where $\bar{\mathcal{P}}_0 = \sigma[\tau, \phi, \alpha]$ and τ, ϕ, α are the type, value, and selection predicate of u_E. We assert that the algorithm $S :=$ `evaluate`$(\bar{\mathcal{P}}, T_D)$ is complete and sound.

Induction proof: The evaluation of $\bar{\mathcal{P}}_0$ yields a set S_0 containing all nodes in T_D that are matches of u_E. The evaluation of $S_0 \mathbin{|\!\bowtie\!\triangleleft} S_i$ yields a set $S_{0,i}$, which contains a subset of S_0 with all nodes that have descendants in S_i. The evaluation of $S := S_{0,1} \sqcap S_{0,2} \sqcap \ldots \sqcap S_{0,m}$ ensures that each node in S has a descendant in each S_i $(1 \leq i \leq m)$. Completeness: The algorithm fulfills the requirements of Definition 6.12 because (i) all matches of u_E are selected, (ii) for each s-child v_{E_i} of u_E $(1 \leq i \leq m)$, the roots of all images of approximate embeddings of $T_E[v_{E_i}]$ are in S_i, and (iii) all paths between matches of u_E and nodes in S_i are found during the joins. Soundness: Let $(u_D, c, \tilde{c})$ be an arbitrary tuple in S. $T_D[u_D]$ directly includes images of approximate embeddings of $T_E[u_E]$ because (i) u_D is a match of u_E and (ii) for each s-child v_{E_i} of u_E $(1 \leq i \leq m)$, a subtree of $T_D[u_D]$ directly includes an image of an approximate embedding of $T_E[v_{E_i}]$. If $(u_D, c, \tilde{c}) \in S$, then $(u_D, c_0, \tilde{c}_0) \in S_0$ exists. The algorithm calculates

$$\begin{aligned} c \;=\; & 0 + c_0 + \min\{\, c_1 + \mathit{nodedist}(u_D, v_{D_1}) \mid (v_{D_1}, c_1, \tilde{c}_1) \in S_1 \wedge u_D \rightsquigarrow v_{D_1} \,\} + \\ & 0 + c_0 + \min\{\, c_2 + \mathit{nodedist}(u_D, v_{D_2}) \mid (v_{D_2}, c_2, \tilde{c}_2) \in S_2 \wedge u_D \rightsquigarrow v_{D_2} \,\} - \tilde{c}_0 + \\ & \ldots \\ & 0 + c_0 + \min\{\, c_m + \mathit{nodedist}(u_D, v_{D_m}) \mid (v_{D_m}, c_m, \tilde{c}_m) \in S_m \wedge u_D \rightsquigarrow v_{D_m} \,\} - \tilde{c}_0. \end{aligned}$$

Because c_0 and $\tilde{c}_0$ are initialized with the same cost, only one summand c_0 remains. For conjunctive queries $c_0 = 0$ holds. The cost c is the cost of the image with the lowest cost

among all images of approximate embeddings of $T_E[u_E]$ directly included in $T_D[u_D]$. Assume there were an image with cost $c' < c$. This may happen for three reasons: (i) Not all images for $T_E[v_{E_i}]$ $(1 \leq i \leq m)$ have been found. (ii) There is a tuple $(v_{D_i}, c_i, \tilde{c}_i) \in S_i$ $(1 \leq i \leq m)$ such that c_i is not the cost of the lowest-cost image of an approximate embedding of $T_E[v_{E_i}]$ directly included in $T_D[v_{D_i}]$. (iii) There is an image of $T_E[u_E]$ directly included in $T_D[u_D]$ with a smaller sum of distances between u_D and the roots of the images of $T_E[v_{E_i}]$ $(1 \leq i \leq m)$. Case (i) and (ii) can be excluded because by assumption S_i $(1 \leq i \leq m)$ contains the roots of all images of $T_E[v_{E_i}]$ and all costs in S_i are minimal. Case (iii) is impossible because the cost calculation follows Definition 6.13 (and is correct according to Lemma 6.3), includes all combinations of images paths, and selects c as the lowest cost of all combinations.

Because the evaluation of the subplan created for each s-subtree of T_E is complete and sound, we conclude that `evaluate`$(\mathcal{P}, T_D)$ is complete and sound, i.e., solves the all-results problem for Q and T_D. □

6.5.2 Boolean Queries

Using the findings from the evaluation of conjunctive queries, we investigate the evaluation of *Boolean queries*, i.e., queries containing both "`and`" and "`or`" operators. We still do not allow permutations, deletions, or value changes. Our theoretical model defines that a Boolean query is transformed to its hierarchical disjunctive normal form (HDNF) (see Section 4.4), and that the conjuncts of the HDNF are mapped to query trees. Each query tree is transformed and evaluated independently. For each logical document that matches any of the transformed query trees, the one with the lowest embedding cost is chosen. A naive method to implement this theoretical model would be the following: We create the HDNF of the query, create the expanded representation of each conjunct, and construct an execution plan for each conjunct. Next, we apply union operators to the result sets. For each set of node-cost pairs that refer to the same data-tree node, the operator keeps only the one with the lowest embedding cost. Therefore, the evaluation of the plan solves the all-results problem. As an example, consider the query

```
cd/title["piano" and ("concerto" or "sonata")]
```

and its HDNF

```
cd/title["piano" and "concerto"] or cd/title["piano" and "sonata"].
```

Figure 6.11(a) shows the two query trees in the separated representation of the query; Figure 6.11(b) shows the expanded representations constructed for the conjuncts of the query. Figure 6.12 depicts the query execution plan with a top-level union operator.

(a) Separated representation

(b) Expanded representations

Figure 6.11: The separated representation and the expanded representations for the query `cd/title["piano" and "concerto"] or cd/title["piano" and "sonata"]`.

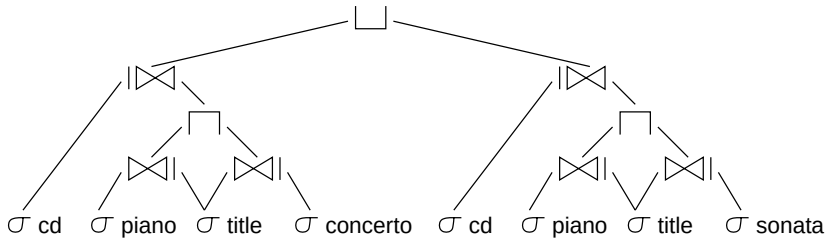
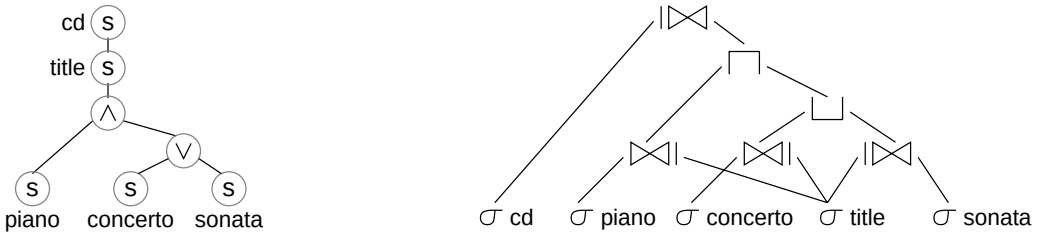


Figure 6.12: The execution plan constructed according to the two expanded representations shown in Figure 6.11(b). For each logical document, the top-level union operator selects the lowest-cost embedding.

Using the operator equivalences of Lemma 6.1 on page 81, the top-level union operator can be "pushed down": The two topmost join operators have the same ancestor set (the cd nodes). Let S_1 be this set and S_2, S_3 be the descendant sets of the left and right joins, respectively. Then, according to equivalence 6.6, the expression $(S_1 \mathbin{|\!\bowtie\!\triangleleft} S_2) \sqcup (S_1 \mathbin{|\!\bowtie\!\triangleleft} S_3)$ can be substituted by $S_1 \mathbin{|\!\bowtie\!\triangleleft} (S_2 \sqcup S_3)$. Because the left operands of the two intersections produce the same result sets, we can apply equivalence 6.5 to factor out the union operator. The plan resulting from the two substitutions is depicted in Figure 6.13(b). This plan is identical to a plan created by Algorithm 6.4.

The following lemma shows that the "in-place" evaluation of Boolean operators is feasible because the execution plans for a query and its counterpart in HDNF are equivalent. Two query-execution plans $\mathcal{P}$ and $\mathcal{P}'$ are equivalent if and only if $\texttt{evaluate}(\mathcal{P}, T_D) = \texttt{evaluate}(\mathcal{P}', T_D)$ for any data tree T_D. A corollary of the lemma is that the evaluation of a query-execution plan constructed by Algorithm 6.4 for a Boolean query *not* in HDNF is complete and sound.

Lemma 6.5 (Plan equivalence) *Let Q be a query, and $Q' = Q'_1 \vee Q'_2 \vee \ldots \vee Q'_n$ be its HDNF. Let $\mathcal{P}$ be the execution plan for Q created by Algorithm 6.4, and $\mathcal{P}' = \mathcal{P}'_1 \sqcup \mathcal{P}'_2 \sqcup \ldots \sqcup \mathcal{P}'_n$ be*



(a) Expanded representation

(b) Execution plan

Figure 6.13: The expanded representation and the execution plan for the query `cd/title["piano" and ("concerto" or "sonata")]`.

the execution plan for Q', where $\mathcal{P}'_j$ is the execution plan for the conjunct Q'_j $(1 \leq j \leq n)$ created by Algorithm 6.4. Then, $\mathcal{P}$ and $\mathcal{P}'$ are equivalent.

Proof: We prove the lemma by induction over the subqueries of Q, in the order they are visited by Algorithm 4.1 on page 49. For each subquery $\bar{Q}$ and its HDNF $\bar{Q}'$, we prove that the subplans $\bar{\mathcal{P}}$ and $\bar{\mathcal{P}}'$ created for $\bar{Q}$ and $\bar{Q}'$ are equivalent. To simplify the query syntax, we write "$\wedge$", "$\vee$" instead of "`and`", "`or`". Also, we omit all operator superscripts that are zero.

Base step: Let $\bar{Q} = \bar{Q}' = s_0$, where s_0 is a selector. Then $\bar{\mathcal{P}}$ and $\bar{\mathcal{P}}'$ are equivalent. Now let $\bar{Q} = s_0[expr(s_1, s_2, \ldots, s_m)]$, where $s_0, s_1, s_2, \ldots, s_m$ are selectors and $expr(s_1, s_2, \ldots, s_m)$ is a Boolean expression. The HDNF of $\bar{Q}$ has the form $\bar{Q}' = s_0[F_1] \vee s_0[F_2] \vee \ldots \vee s_0[F_n]$, where each F_j $(1 \leq j \leq n)$ is a conjunctive formula consisting of some of the selectors $s_1, s_2, \ldots, s_m$. Let $\bar{\mathcal{P}}_i$ be the selection operator created for s_i $(0 \leq i \leq m)$. The subplan created for $\bar{Q}$ has the form $\bar{\mathcal{P}} = expr(\bar{\mathcal{P}}_0 \Vdash\!\bowtie\! \bar{\mathcal{P}}_1, \bar{\mathcal{P}}_0 \Vdash\!\bowtie\! \bar{\mathcal{P}}_2, \ldots, \bar{\mathcal{P}}_0 \Vdash\!\bowtie\! \bar{\mathcal{P}}_m)$, where the join operators are connected by $\sqcap$ and $\sqcup$-operators according to the structure of $\wedge$ and $\vee$-operators in $\bar{Q}$. The subplan for $\bar{Q}'$ has the form $\bar{\mathcal{P}}' = \bar{\mathcal{P}}'_1 \sqcup \bar{\mathcal{P}}'_2 \sqcup \ldots \sqcup \bar{\mathcal{P}}'_n$, where $\bar{\mathcal{P}}'_j = \bar{\mathcal{P}}'_{j,1} \sqcap \bar{\mathcal{P}}'_{j,2} \sqcap \ldots \sqcap \bar{\mathcal{P}}'_{j,q}$ $(1 \leq j \leq n)$. Each $\bar{\mathcal{P}}'_{j,k}$ $(1 \leq k \leq q)$ corresponds to a join $\bar{\mathcal{P}}_0 \Vdash\!\bowtie\! \bar{\mathcal{P}}_i$ $(1 \leq i \leq m)$ in the subplan $\bar{\mathcal{P}}$ created for $\bar{Q}$. Any negation-free Boolean formula can be transformed into its disjunctive normal form (DNF) by successively substituting each formula $F_1 \wedge (F_2 \vee F_3)$ by $(F_1 \wedge F_2) \vee (F_1 \wedge F_3)$ [Sch95]. The expression $expr(s_1, s_2, \ldots, s_m)$ is negation-free, and therefore the transformation is possible. However, the equivalence $F_1 \wedge (F_2 \vee F_3) = (F_1 \wedge F_2) \vee (F_1 \wedge F_3)$ corresponds to the operator equivalence $S_1 \sqcap (S_2 \sqcup S_3) = (S_1 \sqcap S_2) \sqcup (S_1 \sqcap S_3)$ of Lemma 6.1. Because $\bar{Q}$ can be transformed to $\bar{Q}$, the subplan $\bar{\mathcal{P}}$ can be transformed to $\bar{\mathcal{P}}'$ using this operator equivalence. Because each transformation step results in an equivalent subplan, $\bar{\mathcal{P}}$ and $\bar{\mathcal{P}}'$ are equivalent.

Hypothesis: Let $\bar{Q} = s_0[expr(\bar{Q}_1, \bar{Q}_2, \ldots, \bar{Q}_m)]$, where $expr(\bar{Q}_1, \bar{Q}_2, \ldots, \bar{Q}_m)$ is a Boolean expression. For each $\bar{Q}_i$ $(1 \leq i \leq m)$ the HDNF $\bar{Q}'_i = \bar{Q}'_{i,1} \vee \bar{Q}'_{i,2} \vee \ldots \vee \bar{Q}'_{i,p}$ has been created

in an earlier step. Let $\bar{\mathcal{P}}_i$ be the subplan for $\bar{Q}_i$, and $\bar{\mathcal{P}}'_i = \bar{\mathcal{P}}'_{i,1} \sqcup \bar{\mathcal{P}}'_{i,2} \sqcup \ldots \sqcup \bar{\mathcal{P}}'_{i,p}$ be the subplan for $\bar{Q}'_i$. We assume that $\bar{\mathcal{P}}_i$ and $\bar{\mathcal{P}}'_i$ are equivalent ($1 \leq i \leq m$).

Induction step: Let $\bar{Q}'$ be the HDNF of $\bar{Q}$, $\bar{\mathcal{P}}$ be the subplan created for $\bar{Q}$, and $\bar{\mathcal{P}}'$ be the subplan created for $\bar{Q}'$. We assert that $\bar{\mathcal{P}}$ and $\bar{\mathcal{P}}'$ are equivalent.

Induction proof: Let $\bar{Q}_i$ be an arbitrary subquery of $\bar{Q}$ ($1 \leq i \leq m$). The subplan created for $\bar{Q}_i$ has the form $\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}_i$. The subplan created for $\bar{Q}'_i$ has the form $(\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,1}) \sqcup (\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,2}) \sqcup \ldots \sqcup (\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,p})$. By successively applying the operator equivalence 6.6 of Lemma 6.1, the join operator can be factored out. It holds

$$(\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,1}) \sqcup (\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,2}) \sqcup \ldots \sqcup (\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_{i,p}) = \bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft (\bar{\mathcal{P}}'_{i,1} \sqcup \bar{\mathcal{P}}'_{i,2} \sqcup \ldots \sqcup \bar{\mathcal{P}}'_{i,p}) = \bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}'_i.$$

Because $\bar{\mathcal{P}}'_i$ is equivalent to $\bar{\mathcal{P}}_i$, the subplans created for $\bar{Q}_i$ and $\bar{Q}'_i$ are equivalent. Therefore, $\bar{\mathcal{P}}_i$ is a proper subplan for $\bar{Q}_i$, and we can use the symbols $\bar{Q}_i$ and $\bar{Q}'_i$ synonymously. The HDNF of $\bar{Q}$ has the form $\bar{Q}' = s_0[F_1] \vee s_0[F_2] \vee \ldots \vee s_0[F_n]$, where each F_j ($1 \leq j \leq n$) is a conjunctive formula consisting of some of the subqueries $\bar{Q}_1, \bar{Q}_2, \ldots, \bar{Q}_m$ of $\bar{Q}$. Let $\bar{\mathcal{P}}_0$ be the selection operator created for s_0, and $\bar{\mathcal{P}}_i$ be the subplan created for $\bar{Q}_i$ ($1 \leq i \leq m$). The subplan for $\bar{Q}$ has the form $\bar{\mathcal{P}} = expr(\mathcal{P}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}_1, \mathcal{P}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}_2, \ldots, \mathcal{P}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}_m)$, where the join operators are connected by $\sqcap$ and $\sqcup$-operators according to the structure of $\wedge$ and $\vee$-operators in $\bar{Q}$. The subplan for $\bar{Q}'$ has the form $\bar{\mathcal{P}}' = \bar{\mathcal{P}}'_1 \sqcup \bar{\mathcal{P}}'_2 \sqcup \ldots \sqcup \bar{\mathcal{P}}'_n$, where $\bar{\mathcal{P}}'_j = \bar{\mathcal{P}}'_{j,1} \sqcap \bar{\mathcal{P}}'_{j,2} \sqcap \ldots \sqcap \bar{\mathcal{P}}'_{j,q}$ ($1 \leq j \leq n$). Each $\bar{\mathcal{P}}'_{j,k}$ ($1 \leq k \leq q$) corresponds to a join $\bar{\mathcal{P}}_0 \rtimes\!\!\!\triangleleft \bar{\mathcal{P}}_i$ ($1 \leq i \leq m$) in the subplan $\bar{\mathcal{P}}$ created for $\bar{Q}$. $\bar{\mathcal{P}}$ and $\bar{\mathcal{P}}'$ are equivalent because $\bar{\mathcal{P}}$ can be transformed to $\bar{\mathcal{P}}'$ using the operator equivalence $S_1 \sqcap (S_2 \sqcup S_3) = (S_1 \sqcap S_2) \sqcup (S_1 \sqcap S_3)$ of Lemma 6.1 as described for the base step of this proof.

Because the constructed subplans are equivalent for each subquery $\bar{Q}$ of Q and its HDNF $\bar{Q}'$, it follows that $\mathcal{P}$ and $\mathcal{P}'$ are equivalent. □

Corollary 6.1 (Completeness/soundness of Algorithm 6.5 for Boolean queries) *Let Q be a Boolean query for which only insertions are allowed, T_E be the expanded representation of Q, $\mathcal{P}$ be the execution plan created for T_E, and T_D be a data tree. Then algorithm* `evaluate`$(\mathcal{P}, T_D)$ *solves the all-results problem for Q and T_D.*

Proof: Let $Q' = Q'_1 \vee Q'_2 \vee \ldots \vee Q'_n$ be the HDNF of Q. According to Lemma 6.5, $\mathcal{P}$ can be transformed to an equivalent plan $\mathcal{P}' = \mathcal{P}'_1 \sqcup \mathcal{P}'_2 \sqcup \ldots \mathcal{P}'_p$, where $\mathcal{P}'_j$ is the execution plan for the conjunct Q'_j ($1 \leq j \leq n$). According to Lemma 6.4, the evaluation of plans for conjunctive queries is complete and sound. The subplans are connected by $\sqcup$ operators, which select the

lowest-cost embedding for each logical document according to Definition 5.11 on page 62. Therefore, $\mathcal{P}$ is a proper execution plan for Q. It follows that $\texttt{evaluate}(\mathcal{P}, T_D)$ is complete and sound, i.e., solves the all-results problem for Q and T_D. □

6.5.3 Deletions, Permutations, and Value Changes

We distinguish between leaf deletions, which are represented by outerjoins, and deletions of inner nodes, permutations, and value changes, which are represented by unions of alternative subplans. We prove the completeness and soundness of the evaluation algorithm for plans with encoded leaf deletions in a separate lemma, and treat all other transformation the final theorem.

Let v_E be an s-leaf with $\mathsf{delres} \notin \mathit{mod}(v_E)$, and u_E be its s-parent. Algorithm 6.4 on page 87 creates selectors for u_E and v_E, and connects them by an outerjoin operator. For each data-tree node u_D matching u_E, this operator tests whether a descendant of u_D matching v_E exists, and whether the cost of the embedding image of $T_E[u_E]$ is lower if v_E is kept or deleted. The following lemma proves that the evaluation of a plan with encoded deletions of leaves is complete and sound.

Lemma 6.6 (Completeness/soundness of Algorithm 6.5 for leaf deletions) *Let Q be a query for which insertions and deletions of leaves are allowed, T_E be the expanded representation of Q, $\mathcal{P}$ be the execution plan created for T_E, and T_D be a data tree. Then algorithm* $\texttt{evaluate}(\mathcal{P}, T_D)$ *solves the all-results problem for Q and T_D.*

Proof: Let v_E be an arbitrary s-leaf in T_E such that $\mathsf{delres} \notin \mathit{mod}(v_E)$. Let u_E be the s-parent of v_E. Algorithm 6.4 creates a subplan $\bar{\mathcal{P}}_1$ for u_E and a subplan $\bar{\mathcal{P}}_2$ for v_E. The subplan for u_E (v_E) consists of selection operators for the original value of u_E (u_E) and for each of its alternative values. Let S_1 be the set computed by $\bar{\mathcal{P}}_1$, $(u_D, c_1, \tilde{c}_1) \in S_1$) be an arbitrary tuple, and S_2 be the set computed by $\bar{\mathcal{P}}_2$. The outerjoin passes the tuple $(u_D, c, \tilde{c})$ to the result set independently of whether u_D has descendants in S_2 or not. The cost c is calculated as follows:

$$c = c_t + c_1 + \min(c_d, \min\{\, c_2 + \mathit{nodedist}(u_D, v_D) \mid (v_D, c_2, \tilde{c}_2) \in S_2 \wedge u_D \rightsquigarrow v_D\}).$$

The inner "min" operator selects the lowest-cost descendant (see Lemma 6.4 on page 92 for the correctness of this calculation); the outer "min" operator selects the minimum of this cost and the deletion cost c_d of the leaf. Clearly, this is a correct method to select the minimum

cost of keeping or deleting the leaf v_E in the context of the subtree $T_D[u_D]$. The costs c_t and c_1 are added in both cases. Because the same method is used for all tuples in S_1 and for all s-leaves in T_E, the evaluation of a plan with subplans representing leaf deletions is complete and sound. □

In an expanded query representation, deletions of inner nodes and permutations are represented in an analogous manner: A subtree represents the original subquery and another one the transformed subquery. Both subtrees are connected by a $\vee$-node. Consequently, the subplans for deletions and permutations created by Algorithm 6.4 have analogous structures: One subplan represents the original subtree, another one represents the transformed subtree. Both subplans are connected by a union operator. Algorithm 6.4 also uses union operators to connect the selectors created for the alternative values of an s-node. The use of a single primitive for the uniform representation of disjunctions, deletions, permutations, and value changes allows us to interpret a plan created for an expanded representation as a plan created for a Boolean query that represents all transformations as disjunctive subqueries. We make use of this analogy in the final theorem, where we show that a query-execution plan with encoded deletions of inner nodes, permutations, and value changes is complete and sound because all variants are represented by separate subplans connected by union operators. The evaluation of the union operators ensures that the lowest-cost embedding images for each subtree of the data tree are found. In the proof of the theorem, we assume that the encoding of deletions, permutations, and value-changes into an expanded query representation is correct, i.e., all transformed query trees can be derived.

Theorem 6.1 (Completeness/soundness of Algorithm 6.5) *Let Q be a query, T_E be the expanded representation of Q, $\mathcal{P}$ be the execution plan created for T_E, and T_D be a data tree. Then algorithm* `evaluate`$(\mathcal{P}, T_D)$ *solves the all-results problem for Q and T_D.*

Proof: We prove the theorem by induction over the subplans of $\mathcal{P}$ in the order they are created by Algorithm 6.4. This order determines the order of evaluation. Because deletions, permutations, and value-changes are represented by unions of alternative subplans, we only consider subplans of the form $\bar{\mathcal{P}}_1 \sqcup \bar{\mathcal{P}}_2^{c_t}$.

Base step: If $\bar{\mathcal{P}}_1$ and $\bar{\mathcal{P}}_2$ do not contain union operators for the representation of deletions, permutations, and value-changes, then the algorithms `evaluate`$(\bar{\mathcal{P}}_1, T_D)$ and `evaluate`$(\bar{\mathcal{P}}_2, T_D)$ are complete and sound according to Corollary 6.1.

Value change: $\bar{\mathcal{P}}_1$ is a selector created for the original value α of an s-node u_E in T_E; $\bar{\mathcal{P}}_2^{c_t}$ is a selector created for an alternative value α' of α. The superscript c_t is the cost of changing α

to α'. The algorithm $\texttt{evaluate}(\bar{\mathcal{P}}_1 \sqcup \bar{\mathcal{P}}_2^{c_t}, T_D)$ is complete because it selects all nodes in T_D matching u_E with value α or value α'. It is sound because it only selects nodes in T_D matching u_E with values α or α', and because it initializes the cost of each result tuple with either 0 or c_t. These are the correct costs of either using the original value α or its alternative α'.

Deletion: $\bar{\mathcal{P}}_1$ represents a subtree $T_E[u_E]$ containing a node v_E; $\bar{\mathcal{P}}_2^{c_t}$ represents the same subtree without v_E; c_t is the deletion cost of v_E. Let $S_1 = \texttt{evaluate}(\bar{\mathcal{P}}_1, T_D)$ and $S_2 = \texttt{evaluate}(\bar{\mathcal{P}}_2^{c_t}, T_D)$. Let u_D be a node that is either only in S_2, or in both S_1 and S_2 (only in S_1 is impossible). In the first case, the subtree $T_E[u_E]$ has embedding images directly included in $T_D[u_D]$ only if v_E has been deleted. In this case, c_t is added to the primary cost of u_D. Otherwise, the subtree $T_E[u_E]$ including v_E has embedding images. Then, the union operator selects the minimum cost of keeping or deleting v_E. Because the union operator correctly selects the cost of the lowest-cost operand for each pair of tuples in S_1 and S_2, the algorithm $\texttt{evaluate}(\bar{\mathcal{P}}_1 \sqcup \bar{\mathcal{P}}_2^{c_t}, T_D)$ is complete and sound.

Permutation: $\bar{\mathcal{P}}_1$ represents a subtree $T_E[u_E]$ created for the original query; $\bar{\mathcal{P}}_2^{c_t}$ represents a subtree $T_E[v_E]$ where the nodes u_E and v_E have been permuted; c_t is the permutation cost of u_E and v_E. Let S_1 (S_2) be the result of the evaluation of $\bar{\mathcal{P}}_1$ ($\bar{\mathcal{P}}_2^{c_t}$). Let u_D be a node that is either in S_1 or S_2 (both S_1 and S_2 is impossible). If u_D is in S_1, then the union operator passes it to the result set without changing its cost. Otherwise, the cost of u_D is increased by c_t. Because the union operator correctly selects the cost of the lowest-cost operand for each pair of tuples in S_1 and S_2, the algorithm $\texttt{evaluate}(\bar{\mathcal{P}}_1 \sqcup \bar{\mathcal{P}}_2^{c_t}, T_D)$ is complete and sound.

Hypothesis: $\bar{\mathcal{P}}_1$ and $\bar{\mathcal{P}}_2^{c_t}$ are arbitrary subplans of $\mathcal{P}$ such that the algorithms $\texttt{evaluate}(\bar{\mathcal{P}}_1, T_D)$ and $\texttt{evaluate}(\bar{\mathcal{P}}_2, T_D)$ are complete and sound.

Induction step: We assert that the algorithm $\texttt{evaluate}(\bar{\mathcal{P}}_1 \sqcup \bar{\mathcal{P}}_2^{c_t}, T_D)$ is complete and sound.

Induction proof: $\bar{\mathcal{P}}_1$ is an arbitrary subplan; $\bar{\mathcal{P}}_2^{c_t}$ is either a selector created for an alternative value an s-node, a subplan representing the deletion of an inner s-node, or a subplan representing the permutation of two nodes.

Value change: $\bar{\mathcal{P}}_1$ consists of union operators that connect selectors created for the original value of an s-node u_E in T_E and the first k alternative values. $\bar{\mathcal{P}}_2^{c_t}$ is a selectors created for the $(k+1)$th alternative value α' of u_E. The algorithm $\texttt{evaluate}(\bar{\mathcal{P}}, T_D)$ is complete because it passes all tuples computed by $\bar{\mathcal{P}}_1$ to the result set and adds all matches for u_E with the value α'. It is sound because it adds only matches for u_E with the value α' to the result set. If a result tuple comes from $\bar{\mathcal{P}}_1$, then its cost is correct by assumption. Otherwise, the cost is initialized with c_t, which is the correct cost of changing the value of u_E from α to α'.

Deletion, permutation: Same argumentation as in the base step of this proof.

Because the evaluation of each subplan of $\mathcal{P}$ is complete and sound, we conclude that the algorithm `evaluate`$(\mathcal{P}, T_D)$ is complete and sound, i.e., it solves the all-results problem for Q and T_D. □

6.6 Optimizing Direct Query Evaluation

In this section, we present two techniques for optimizing the evaluation of query-execution plans. First, we make use of operator equivalences to compact plans. Compacted plans can be evaluated more efficiently because less operators have to be executed. Second, we use dynamic programming to avoid the repeated evaluation of shared subplans.

6.6.1 Compacting Query-Execution Plans

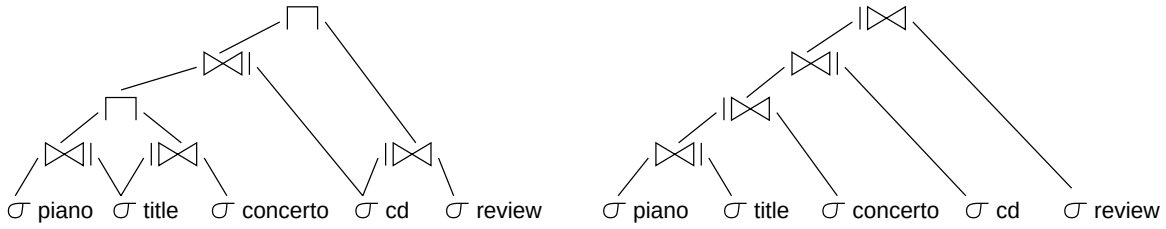
The query-execution plans created by Algorithm 6.4 on page 87 all have the same general structure: At the bottom level are selection operators; at the next level are join operators; and at the remaining levels are union and intersection operators. Many intersections combine the results of two join operators that check the same set of ancestors against different sets of descendants. The same effect yields a sequence of two join operators, where the output of the first operator is used as the ancestor set of the second one. This relationship is captured by the operator equivalence

$$(S_1 \mathbin{\text{I}\!\bowtie\!\text{I}} S_2) \mathbin{\text{I}\!\bowtie\!\text{I}} S_3 = (S_1 \mathbin{\text{I}\!\bowtie\!\text{I}} S_2) \sqcap (S_1 \mathbin{\text{I}\!\bowtie\!\text{I}} S_3)$$

of Lemma 6.1 on page 81, provided that for each node in set S_1 the cost and the backup cost are equal. This precondition holds because the tuples in S_1 are created by a selection operator, which initializes the cost and the backup cost with the same value (see Definition 6.5 on page 78). Using this equivalence, we can transform the plan depicted in Figure 6.14(a) on the next page to the compacted plan shown in Figure 6.14(b), in which all intersections are eliminated.

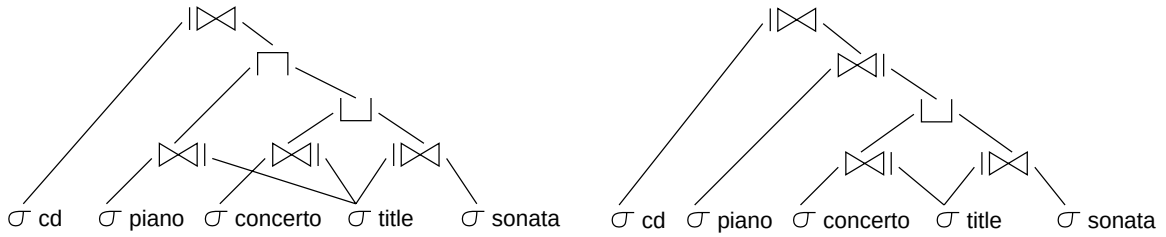
In fact, intersections are *only* needed if the query contains Boolean formulae that are not in disjunctive normal form. In many cases, we can discard the intersections even then. If we can guarantee that the cost and the backup cost are equal for each node in a set S_1, and the nodes in S_1 are a superset of the nodes in another set S_3, then the equivalence

$$(S_1 \mathbin{\text{I}\!\bowtie\!\text{I}} S_2) \sqcap S_3 = S_3 \mathbin{\text{I}\!\bowtie\!\text{I}} S_2$$



(a) Original plan (b) Compacted plan

Figure 6.14: Original and compacted execution plan for the query `cd[title["piano" and "concerto"] and review]`.



(a) Original plan (b) Compacted plan

Figure 6.15: Original and compacted execution plan for the query `cd/title["piano" and ("concerto" or "sonata")]`.

of Lemma 6.1 holds. In this formula, S_2 is an arbitrary set of node-cost tuples. We can use this equivalence to simplify, e.g., the plan depicted in Figure 6.15(a). Figure 6.15(b) shows the result of the transformation.

The simplest way of compacting query-execution plans is the merging of duplicate subplans. In plans constructed by Algorithm 6.4, only the selection operators (or unions of them) created for inner s-nodes are shared. In many cases, however, additional subplans can be merged. Consider the plan depicted in Figure 6.9 on page 85 where the subplans $\sigma\,\mathsf{concerto} \sqcup \sigma^3\,\mathsf{sonata}$ and $\sigma\,\mathsf{title}$ appear twice. By merging these subplans and removing the intersections, we get the compacted plan depicted in Figure 6.16 on the facing page.

Besides of the discussed cases, several other restructurings based on operator equivalences are possible. Particularly the disjunctive laws of Lemma 6.1 help to further reduce the number of operators. Because the application of these equivalences is obvious, we omit further details.

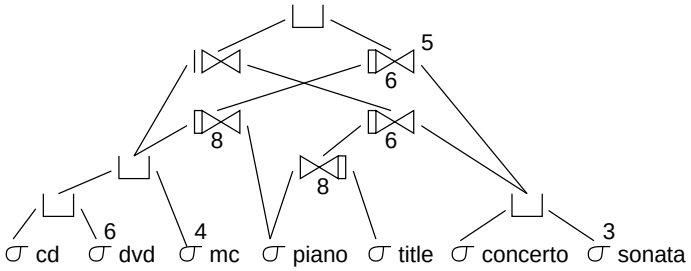


Figure 6.16: Compacted and merged version of the plan shown in Figure 6.9 on page 85.

6.6.2 Dynamic Programming

Dynamic programming is an algorithmic technique in which an optimization problem is solved by caching subproblem solutions rather than recomputing them. We apply the dynamic-programming principle to the evaluation of a query-execution plan by caching the result sets computed for common subplans. Consider Figure 6.16, where three subplans are shared (they each have more than one incoming edge). The plan-evaluation algorithm presented in Section 6.4 evaluates each of these subplans twice. By caching the results computed during the first pass, the evaluation of the plan can be significantly accelerated. Because almost every plan contains shared subplans (as discussed in the previous subsection), dynamic programming is a powerful means to optimize the evaluation of query-execution plans.

To prepare a query-execution plan for use in a dynamic-programming algorithm, we assign a unique number to each node with more than one parent. We use the notation $id(v_\mathcal{P})$ to refer to the number of node $v_\mathcal{P}$. Algorithm 6.6 on the next page shows an improved version of Algorithm 6.5 on page 88 that uses dynamic programming. It accepts an additional parameter H, which denotes a hash table used for storing intermediate results. The function `evaluate` traverses the plan top-down, and executes the operators during its ascent. If a number $id(v_\mathcal{P})$ for the current node $v_\mathcal{P}$ is defined, and a hash-table entry for that number exists, then the evaluation of the subplan ends and the results are returned (Lines 1 and 2). Otherwise, the plan is traversed and the operators are executed (Lines 3–13). This part of the algorithm is identical to the simple plan-evaluation implemented by Algorithm 6.5. If the current subplan is shared, then the results of the evaluation are stored in the hash table (Lines 14 and 15).

Algorithm 6.6 evaluates a query-execution plan based on dynamic programming.

function `evaluate`$(\mathcal{P}, v_\mathcal{P}, T_D, H)$

params: $\mathcal{P} = (N, E, r, op, param, delcost, transcost)$ – a query-execution plan,
$v_\mathcal{P}$ – a node in N,
T_D – a data tree,
H – a hash map of intermediate results,

returns: S – the results of the subplan of $\mathcal{P}$ with root $v_\mathcal{P}$.

1: **if** $id(v_\mathcal{P})$ is defined and $H[id(v_\mathcal{P})]$ exists **then**
2: **return** $H[id(v_\mathcal{P})]$
3: $c_t := transcost(v_\mathcal{P})$
4: **if** $op(v_\mathcal{P}) = \sigma$ **then**
5: $S := \sigma^{c_t}[param(v_\mathcal{P})]\ T_D$
6: **else**
7: $S_1 :=$ `evaluate`$(\mathcal{P}, child_1(v_\mathcal{P}), T_D, H)$
8: $S_2 :=$ `evaluate`$(\mathcal{P}, child_2(v_\mathcal{P}), T_D, H)$
9: **if** $op(v_\mathcal{P}) = \rhd\!\lhd$ **then**
10: $c_d := delcost(v_\mathcal{P})$
11: $S := S_1\ \rhd\!\lhd_{c_d}^{c_t}\ S_2$
12: **else**
13: $S := S_1\ op(v_\mathcal{P})^{c_t}\ S_2$
14: **if** $id(v_\mathcal{P})$ is defined **then**
15: $H[id(v_\mathcal{P})] := S$
16: **return** S

6.7 Space Complexity of Query-Execution Plans

The space complexity of a query-execution plan describes the maximum number of operators that it can consist of, given the number of query selectors, the number of permitted value changes per selector, and the number of permitted node permutations. We do not investigate the time and space complexity of the *evaluation* of a plan here. We postpone this analysis to Section 8.5, once we will have introduced the algorithms that implement the plan operators. The following list summarizes the symbols that we will use in the complexity formulae:

d – maximum number of children of a query selector (query degree),
n – number of query selectors (query size),
p – number of permitted permutations for the query,
v – maximum number of permitted value changes per query selector.

For the complexity analysis, we assume that identical subplans have been merged as described in the previous section. This particularly lowers the space complexity of plans with encoded deletions and permutations, where alternative subplans have common parts.

Boolean queries. Each selector in a Boolean query (allowing insertions only) is represented by a selection operator. Each selector (except the root) is connected to its parent selector by a join operator, and a union or intersection operator. Therefore, a query-execution plan for a Boolean query consists of $O(n)$ operators.

Deletions. Each permitted deletion of a leaf changes a join operator to an outerjoin operator. The size of the plan does not increase. The deletion of an inner selector is represented by two alternative subplans $\mathcal{P}_1$ and $\mathcal{P}_2$ connected by a union operator. $\mathcal{P}_1$ and $\mathcal{P}_2$ share $O(d)$ identical sublans constructed for the children of the selector. However, the remaining operators in $\mathcal{P}_1$ and $\mathcal{P}_2$ represent different Boolean expressions in the context of different parent selectors. Therefore, a plan with one encoded deletion has $O(d)$ additional join operators, and $O(d)$ additional union and intersection operators. A plan that encodes all deletions consists of $O(n \cdot d)$ operators.

Permutations. A permutation of two selectors is represented by the two alternative subplans $\mathcal{P}_1$ and $\mathcal{P}_2$ connected by a union operator. $\mathcal{P}_1$ and $\mathcal{P}_2$ share $O(d)$ identical sublans constructed for the children of the first selector, and $O(d)$ identical sublans for the children of the second

selector. $O(d)$ additional union and intersection operators are necessary to represent the Boolean expressions in the permuted parts of the query. A plan that encodes p permutations consists of $O(n + p \cdot d)$ operators.

Value changes. The original value of a query selector is represented by a selection operator. Each of its $O(v)$ alternative values is represented by a selection operator and a union operator. Because indentical subplans are merged, each selector appears only once, even if deletions and permutations are encoded. Therefore, a plan that encodes all permitted value-changes per selector consists of $O(n \cdot v)$ operators.

If permutations, deletions, and value changes are encoded together in a plan, then its total number of operators is

$$O((n \cdot v + p) \cdot d).$$

In typical cases, the parameters n (number of query selectors) and d (query degree) are small numbers. The parameters v and p are determined during the configuration of the plan-generation algorithm, and do not depend on query and data properties. Therefore, the size of a plan can be effectively controlled by choosing appropriated values for v and p, balancing matching flexibility and evaluation efficiency.

6.8 Related Work

The assembling of query-execution plans from a fixed set of operators is a standard technique for the implementation of query processors for relational and object-oriented database management systems. This technique is also used in some query processors for semistructured data and XML, e.g., in the query processors of Lore [MW99] and Xyleme [ACVW00].

Most XML query languages support some kind of regular path expressions (see Section 2.1). In particular, they often have operators to "skip" nodes on a path (e.g, the "//" operator in XPath [CD99]). To implement this language primitive in a query processor, several researchers proposed ancestor-descendant joins [Sch01a, ZND+01, ACS02, AKJK+02, CVZ+02], adopting older ideas developed for text databases [MZ92, Nav95, CCB95a].

Ancestor-descendant joins that measure the distances between nodes have been independently proposed by Amer-Yahia et. al. [ACS02] and by ourselves [Sch01a, Sch02a]. Amer-Yahia et. al. introduce join and outerjoin operators, and show how query-execution plans for approximate tree-pattern queries can be built with these operators. Their join operators differ from ours in

two aspects: First, they construct "full" relations consisting of all ancestor-descendant pairs found in the data tree. Second, they use a node-distance measure that is defined as a function of the number of nodes on the path between an ancestor and a descendant. In contrast, our joins only pass ancestor nodes to the result set and calculate the lowest cost of all descendants for each ancestor. The node distance as one constituent of the lowest cost is defined as the sum of insertion costs of all nodes on the path between an ancestor and a descendant. In addition to joins and outerjoins, we use cost-calculating selections, unions, and intersections as parts of an operator algebra. The union operator allows the implementation of disjunctions, deletions, permutations, and value changes in a uniform manner. In contrast, the approach proposed by Amer-Yahia et. al. is limited to conjunctive queries. Node generalizations (as restricted variants of value changes) and deletions are represented by means of join predicates; permutations are not supported.