Chapter 8

Efficient Algorithms for Plan Operators

In the previous two chapters, we defined five operators that perform operations on sets of node-cost tuples. In this chapter, we present algorithms and associated data structures for implementing these operators. We adopt a well-known numbering scheme that allows to test whether two nodes in a tree have an ancestor-descendant relationship. We extend this scheme in such a way that it allows to calculate the sum of the insertion costs of nodes on a path, without actually traversing the path. Because the numbers assigned to the nodes provide all of the information required for verifying the relationships between nodes in a tree, we can finally ignore the tree itself. Instead, we use lists in which each entry stores the numbers assigned to a particular node. We show how standard indexes like inverted files or B-trees can be used to initialize lists so that they store information about all nodes that have a given type and fulfill a given selection predicate. The list-oriented view of trees allows very efficient algorithms for the implementation of our operators: The selection operator makes use of the indexes to initialize the lists passed to other plan operators. The join and the outerjoin operators establish the ancestor-descendant relationships between nodes stored in the operand lists. The union operator merges the operand lists and adds the costs of corresponding nodes; the intersection operator intersects the lists and calculates the minimum of the costs. All operators compute the results in almost linear time with respect to the lengths of the operand lists.

The chapter is organized as follows: In Section 8.1, we review and extend a numbering scheme for trees. In Section 8.2, we show how the numbers for a particular selection of nodes can be retrieved from a standard index. The numbers assigned to nodes are collected in lists, which we introduce in Section 8.3. In Section 8.4, we present the algorithms for the operators, which make use of node information stored in lists. In Section 8.5, we analyze the complexity

bounds of the algorithms and also of the query-execution plans assembled from the operators. Finally, in Section 8.6, we review related work. Throughout the chapter, we use the term *target tree* if the introduced concepts are applicable for both data trees and path trees.

8.1 Compacting and Encoding a Target Tree

The compacting and encoding of a target tree is an intermediate step during the creation of index structures. Note that in practice, the explicit construction (and subsequent compacting and encoding) of a target tree is not necessary, because the encoding of the nodes added to the indexes can be successively constructed during the import of XML documents.

In a compacted target tree, sibling leaves with equal types are merged into a single leaf. Each merged leaf is annotated with a set consisting of the values of the original leaves. The merging of leaves is admissible for the following reasons: First, all merged leaves are cost-equivalent, which means that they have the same distances from each of their ancestors. Second, an embedding is a function that is not injective. Therefore, sibling leaves of a query tree can be mapped to a single leaf of the target tree, provided that the target leaf is annotated with the values of both query leaves. Consequently, each query tree that can be embedded into a target tree can also be embedded into its compacted version.

Given a compacted target tree, we adopt the preorder-bound numbering scheme of trees, which has proven to be useful for the evaluation of containment queries [Nav95, ZND$^+$01]. The numbering scheme is based on nested regions within a continuous text. Each region is identified by its start and end offsets within the text. The application for XML data is straightforward: Each element delimited by a start-tag and an end-tag defines a region; nested elements define nested regions. There is a simple correspondence between text regions and nodes in the target tree: Each inner node u spans a region consisting of the leaves of the subtree rooted at u. Nested subtrees represent nested elements, and therefore represent nested regions. The start offset of a region is equal to the preorder number of the node u covering the region; the end offset (called the *bound*) of the region is equal to the preorder number of the rightmost leaf of the subtree rooted at u. We use the notations $pre(u)$ to refer to the preorder number of u and $bound(u)$ to refer to the region bound defined by u. Given the nodes u and v, u is an ancestor of v if and only if

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

holds. We extend this basic numbering scheme to determine the *node distances* in constant time. Recall from Definition 6.6 on page 79 that the distance between two nodes u and v is

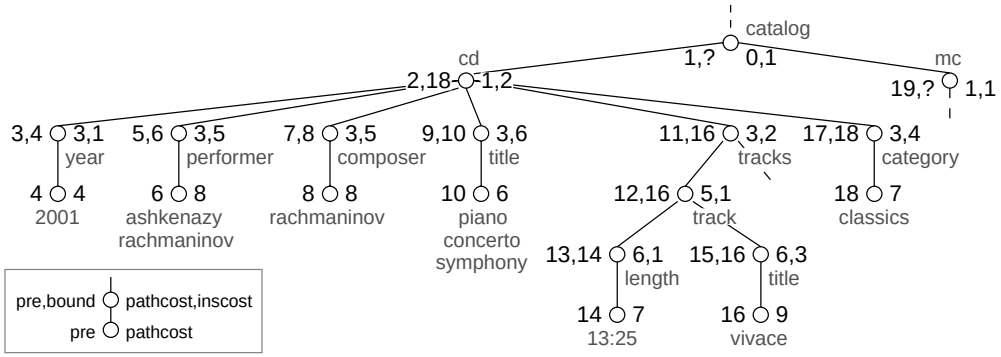


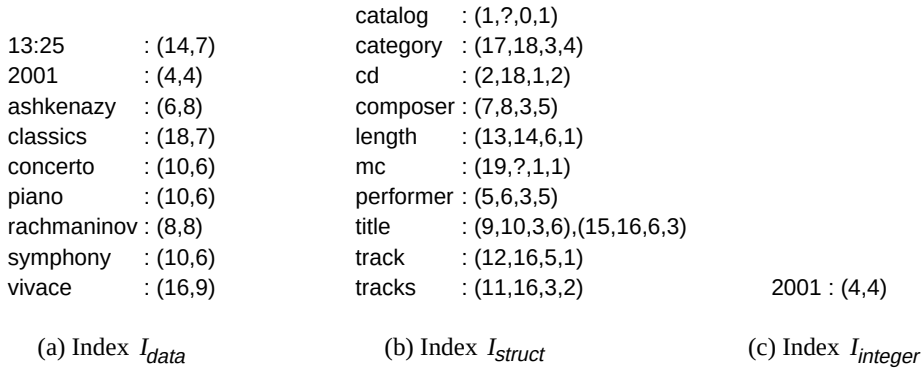
Figure 8.1: The compacted and encoded version of the path tree depicted in Figure 7.2 on page 111.

defined as the sum of the insertion costs of all nodes on the path from u to v (excluding u and v). In order to calculate the node distances, we assign the *path cost* to each node u, which is the total insertion cost of all nodes on the path from the root of the target tree to u (excluding u). We write $pathcost(u)$ to refer to this cost. Furthermore, we define the notation $inscost(u)$ to refer to the insertion cost assigned to u. If u is an ancestor of v, then the node distance between u and v is

$$nodedist(u, v) = pathcost(v) - pathcost(u) - inscost(u).$$

If u is not an ancestor of v, then the distance between the nodes is infinite. The preorder numbers, path costs, and insertion costs are assigned during a depth-first traversal of the target tree; the bounds are assigned bottom-up. If u is a leaf, then the bound is equal to the preorder number, and can be omitted. The insertion cost of a leaf is not needed to for the calculation of the distances, and is therefore omitted as well.

An example of an encoded path tree is shown in Figure 8.1. The preorder number and the bound are assigned to the left side of each node; the path cost and the insertion cost are located at the right side. For simplicity, we omit the node types in the figure. Consider the nodes 11 (tracks) and 16 (vivace). The numbering scheme allows the verification that node 11 is an ancestor of node 16, because $11 < 16 \wedge 16 \geq 16$ evaluates to true. The calculation $9 - 3 - 2 = 4$ yields the sum of the insertion costs of the nodes 12 and 15, and therefore also the distance between the nodes 11 and 16.



(a) Index I_{data} (b) Index I_{struct} (c) Index $I_{integer}$

Figure 8.2: The indexes of the path tree depicted in Figure 8.1 on the page before.

8.2 The Indexes of a Target Tree

To access all nodes of a given type and value, we use type-dependent indexes for both data trees (to evaluate a query using the direct method) and path trees (to construct second-level queries).

A type-dependent index I_τ maps the values of nodes with type τ to *postings*. Each posting entry represents a target-tree node and contains the numbers assigned to the respective node. That is, an entry in a structural index is a quadruple

$$(pre, bound, pathcost, inscost),$$

where the components refer to the corresponding values of the node represented by the entry. An entry in an index for accessing nodes of type *data* is a pair

$$(pre, pathcost),$$

consisting of the preorder number and the path cost of the represented node. Figure 8.2 shows three indexes constructed for the compacted path tree depicted in Figure 8.1 on the preceding page. The numbers are *only* stored within the posting entries; the target tree itself is not necessary for the evaluation of queries. We support two variants for the indexing of type hierarchies:

Replication: Entries belonging to nodes of type τ are stored in the index I_τ, and also in all indexes $I_{\tau'}$, where τ' is a supertype of τ.

Filtering: Entries belonging to nodes of type τ are only stored in the index I_τ, where τ' is a supertype of τ. To detect the actual type of the node, we add additional type information to the entry representing the node.

Any combination of replication and filtering is possible. Our example in Figure 8.2 uses the following combination: The index I_{struct} stores entries representing *element* nodes as well as entries representing nodes of type *attribute* (for simplicity, we omit the type information). In contrast, *integer* nodes are replicated so that they appear in both I_{data} and $I_{integer}$.

The proposed indexing model can be implemented on top of any standard index, as long as the index supports the selection predicates defined by the node type. For instance, a hash table can be used if only equality tests are defined for a type. An example is the index I_{struct} depicted in Figure 8.2(b). For numerical data, where less-than and greater-than must be supported, a B-tree may be used. An example is the index $I_{integer}$ depicted in Figure 8.2(c). Moreover, for text data we may need to support phrase and substring matching, regular expressions, and fault-tolerant string matching based on the edit distance for strings. In these cases, we may implement the index as a suffix tree or suffix array [MM90].

8.3 Lists

The operators defined in Section 6.3 and extended in Section 7.2 work on sets. A set stores information about data-tree or path-tree nodes of a given type and value. Each basic set entry has the structure $(u_D, c, \tilde{c})$, where u_D is a data-tree node, c is the primary cost of u_D (the cost of embedding a query subtree into the data subtree rooted at u_D), and $\tilde{c}$ is the backup cost of c used to compensate the effect of passing c along different paths.

In this section, we show how sets of node-cost tuples can be implemented so that the operators join, outerjoin, union, and intersect are efficient: First, we replace the node component of a tuple by four integers, which are sufficient to describe all necessary node properties. Second, we use *lists* instead of sets. The tuples in the lists are sorted by the topology of the nodes they represent. This allows the implementation of operators that have *linear* time complexity with respect to the lengths of the operand lists.

A *basic list entry* used by the operators of the direct evaluation method implements a basic node-cost tuple $(u_D, c, \tilde{c})$. It has the following structure:

$$(pre, bound, pathcost, inscost, embcost, backcost),$$

where the first four integers store fixed information about the data-tree node u_D. These values are initialized from the corresponding integers assigned to the node (see Section 8.1). The component *embcost* is the primary cost c; the component *backcost* is the backup cost $\tilde{c}$.

For the two schema-driven evaluation methods, we additionally have to track the images of path-tree embeddings, which are later used as second level queries. In analogy to extended node-cost tuples introduced in Section 7.2.1, we propose *extended list entries*, which have the following structure:

$$(pre, bound, pathcost, inscost, embcost, backcost, pointers).$$

The first six components form a basic list entry. The component *pointers* is a set of references to extended list entries. It implements the component N of an extended node-cost tuple $(u_P, c, \tilde{c}, N)$ (see Definition 7.5 on page 115). If u_P is a match of a query node u_Q, then N stores a combination of tuples that represent path-tree nodes matched by the children of u_Q.

Each list is sorted by the preorder numbers of the list entries ascendingly, which reflects the topology of the nodes. If one of the schema-driven evaluation methods is used, the lists consist of k-segments (see Section 7.2.2), where all entries in a particular k-segment represent embedding images of the *same* query subtree in the *same* subtree of the path tree. This means that all entries in a k-segment have the same preorder number, but different embedding costs and pointer sets. List segments are sorted by increasing embedding cost. All operations on lists ensure both the primary sort order (by node number) and the secondary sort order (by embedding cost) of the result list. We define the following notations:

$[\,]$	the empty list,
$\|L\|$	the number of entries in list L,
$L[i]$	the ith entry in L $(1 \leq i \leq \|L\|)$,
$L[i, j]$	the interval of entries limited by the indices i and j.

We use a functional notation to refer to the components of entries. For example, we write $embcost(L[i])$ to access the component *embcost* of the entry $L[i]$. In the algorithms introduced in the following section, we sometimes say that "entry $L_2[j]$ is a descendant of entry $L_1[i]$". This is an abbreviation for

$$pre(L_1[i]) < pre(L_2[j]) \wedge bound(L_1[i]) \geq pre(L_2[j]).$$

8.4 Operations on Lists

In this section, we present algorithms for implementing the five operators defined in Section 6.3. For simplicity, we only present the algorithms for lists with basic entries, which are

used by the direct query-evaluation method (see Chapter 6). The algorithms for lists with extended entries and k-segments used by the schema-driven query-evaluation methods (see Chapter 7) are straightforward extensions.

Each operator is implemented as a function that returns a list. The following table shows the names of the functions.

operator symbol:	σ	$\rtimes$	$\bowtie$	$\sqcup$	$\sqcap$
function name:	`select`	`join`	`outerjoin`	`union`	`intersect`

In the following subsections, we always review the purpose of an operator before we present its list-based implementation.

8.4.1 Selection

The selection operator $\sigma^{c_t}[\tau, \phi, \alpha]$ (see Definition 6.5 on page 78) creates a set of node-cost tuples consisting of all data-tree nodes that have type τ or a subtype of τ, and that fulfill the selection predicate ϕ with respect to value α. The cost component of the tuples is initialized with the transformation cost c_t.

The list-based version of the selection operator (see Algorithm 8.1 on the following page) creates a list of basic entries, where each entry represents a selected data-tree node. After setting up a new list (Line 1), the algorithm locates an index that supports the type τ and the selection predicate ϕ (Line 2). For example, if $\tau = \mathit{integer}$ and $\phi = '\leq'$, then I_τ cannot be a hash table. If the algorithm cannot find an appropriate index, it returns the empty list (Line 3). Otherwise, the algorithm initializes a new list from the posting retrieved from the index. It distinguishes between structural nodes (Lines 4–6), of which the indexes store four integers, and nodes of type *data*, where pairs are stored (Lines 7–9). For each list entry, the values *pre* and *pathcost* are copied from the index entries. If the selected nodes are structural nodes, then the values *bound* and *inscost* are copied as well; otherwise, they are set to zero. The primary cost and the backup cost of each list entry are set to the transformation cost c_t. The list returned by the operator is sorted by the preorder node numbers in ascending order.

8.4.2 Join and Outerjoin

The join operator $S_1 \rtimes^{c_t} S_2$ (see Definition 6.7 on page 79) selects all tuples from the set S_1 representing nodes with descendants in the set S_2. For each ancestor and for each of its

Algorithm 8.1 accesses the index and initializes a list.

function `select`(τ, ϕ, α, c)

params: τ – a node type,
ϕ – a selection condition,
α – a value,
c – a transformation cost,
returns: L – a list initialized from the index.

1: $L := [\,]$
2: Select an index $I_{\tau'}$ such that $\tau' \preceq \tau$ and $I_{\tau'}$ is able to verify ϕ
3: **if** $I_{\tau'}$ does not exist **then return** L
4: **if** $struct \prec \tau$ **then**
5: **for each** tuple $(pre, bound, pathcost, inscost)$ of $I_\tau(\phi, \alpha)$ **do**
6: append $(pre, bound, pathcost, inscost, \alpha, c, c)$ to L
7: **else if** $data \prec \tau$ **then**
8: **for each** tuple $(pre, pathcost)$ of $I_\tau(\phi, \alpha)$ **do**
9: append $(pre, 0, pathcost, 0, \alpha, c, c)$ to L
10: **return** L

descendants, it calculates the sum of the embedding cost assigned to the ancestor, the embedding cost assigned to the descendant, and the node distance between the ancestor and the descendant. From all descendants of the same ancestor, the operator chooses the one with the lowest cost and inserts a new tuple consisting of the ancestor and the updated cost into the set of results. The transformation cost c_t is added to the cost component of each result tuple.

The node-distance function (see Definition 6.6 on page 79) used by the join operator can be easily rewritten to work with list entries. Let $L_A[i]$ be an entry in the ancestor list and $L_D[j]$ be an entry in the descendant list. The distance between the nodes referred to by $L_A[i]$ and $L_D[j]$ is

$$nodedist(L_A[i], L_D[j]) = pathcost(L_D[j]) - pathcost(L_A[i]) - inscost(L_A[i]).$$

Using this function, we can calculate the new embedding cost $embcost(L_A[i])'$ of the node represented by $L_A[i]$ with respect to the node represented by $L_D[j]$ as follows:

$$embcost(L_A[i])' = embcost(L_A[i]) + embcost(L_D[j]) + nodedist(L_A[i], L_D[j]) + c_t,$$

where c_t is a deletion or permutation cost passed to the operator.

Recall that all lists are ascendingly sorted by the preorder node numbers. The sort order reflects the topological relationships of the nodes represented by the list entries: Let $L_D[j]$ and $L_D[k]$ be entries in the list L_D. If $j < k$, then the node represented by $L_D[k]$ is either in

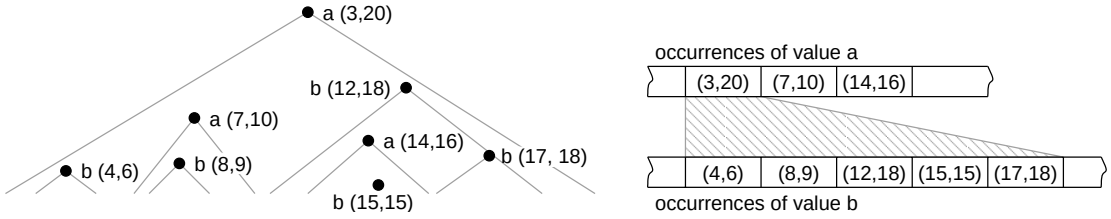
the subtree rooted at the node represented by $L_D[j]$, or it is to the right of this subtree. If both $L_D[j]$ and $L_D[k]$ are descendants of the same entry $L_A[i]$, then all entries within the list interval $L_D[j,k]$ are descendants of $L_A[i]$ as well. If $L_D[j]$ is the leftmost entry in L_D that is a descendant of $L_A[i]$ and $L_D[k]$ is the rightmost entry with that property, then all descendants of $L_A[i]$ in L_D are in the single interval $L_D[j,k]$. An equivalent interpretation is that the node represented by $L_A[i]$ covers a *region*, and all nodes of that region having the same type and value are represented in a single interval of L_D. Therefore, the lowest embedding cost of the node represented by $L_A[i]$ with respect to its descendants in L_D can be calculated as follows:

$$\mathit{embcost}(L_A[i])' = \mathit{embcost}(L_A[i]) + \min\{\mathit{embcost}(L_D[l]) + \mathit{nodedist}(L_A[i], L_D[l]) \mid i \leq l \leq k\} + c_t.$$

For the moment, we assume that the data tree is not recursive, i.e., any pair of type and value appears only once on a path. This assumption implies that the nodes represented by the entries in L_A cover disjoint regions. Moreover, the region covered by the node represented by the entry $L_A[i+1]$ is to the right of the region covered by the node represented by $L_A[i]$. Regarding the particular list L_D, this means that the interval of descendants of $L_A[i+1]$ is to the right of the interval of descendants of $L_A[i]$. The disjointedness of the intervals allows to establish the ancestor-descendant relationship by simultaneously iterating through the lists L_A and L_D. Linear-time joins based on regions are a well-known technique for the implementation of containment queries (see, e.g., [Nav95]).

We now consider the general case of recursive trees, and introduce a join algorithm that works correctly in the case of recursive trees, but still has linear time complexity in case of non-recursive trees. The modification of the algorithm relies on a simple observation concerning the positions of the descendants of a node in a *single* list: Let $L_A[i]$ be an entry in L_A representing node u_D. If u_D has a descendant that has the same type and value as u_D, then this descendant must be represented by an entry in L_A. Moreover, because the data tree is preorder enumerated, and because L_A is sorted all m descendants of u_D in L_A must be in the interval $L_A[i+1, i+m]$.

We now consider the positions of descendants in two *different* lists: Assume that we have already found the interval $L_D[j,k]$ of descendants of $L_A[i]$. If the nodes represented by entries in the interval $L_A[i+1, i+m]$ have descendants in L_D, then those descendants must be among the entries in the interval $L_D[j,k]$. Figure 8.3 on the next page illustrates this observation: All b-valued descendants of node 3 are in a contiguous interval of the list that stores all of the occurrences of b. If we already know that interval, then we also know that all a-valued descendants of node 3 (nodes 7 and 14) have its b-valued descendants in the same interval as node 3. We can restrict the search for descendants to the computed intervals if we observe



(a) Part of a recursive data tree.

(b) Lists with nested intervals.

Figure 8.3: A part of a recursive data tree and the corresponding sections in the lists belonging to the values a and b, respectively. The distance values are not shown.

recursive uses of values on a path. The join operator uses recursive calls to perform the interval search.

Algorithm 8.2 on the facing page shows the implementation of the join operator based on list intervals. The outer loop iterates through the entries in L_A (Line 2). At Lines 3 and 4, the algorithm searches the starting position of an interval of descendants of $L_A[i]$ in L_D. The variable i' marks the first index of that interval (Line 5). The algorithm then searches in the interval of descendants for the one with the smallest sum of embedding costs and node distance (Lines 6–8). If the calculated cost is not infinite, the algorithm creates a copy of the entry $L_A[i]$, updates its embedding cost, and appends the entry to the result list (Lines 9 and 10). At Lines 12–15, recursive subtrees are taken into account. The algorithm searches the first entry in L_A that is *not* a descendant of $L_A[i]$ (Lines 12 and 13). If $L_A[i]$ does not have any descendants in L_D, then all its descendants in L_A cannot have descendants in L_D either. Otherwise, if $L_A[i]$ has both descendants in L_A and in L_D, then the algorithm recursively calls the operator for the computed intervals $L_A[i', i]$ and $L_D[j', j]$ (Lines 14 and 15).

The result list of a join contains only entries from L_A that have descendants in L_D. The outerjoin operator additionally keeps all entries from the ancestor list that do not have descendants. The parameter c_d specifies the deletion cost of the query leaf whose matches are in L_D. If c_d is not infinite, then the new operator copies each entry in the ancestor list L_A to the result list, even if it has no descendants in L_D. In this case, the sum of c_d and c_t is used as the embedding cost of the new entry appended to the result list. To modify the join operator to an outer join, we simply have to insert the expression

$$c_{min} := \min(c_{min}, c_d)$$

between Lines 8 and 9.

Algorithm 8.2 performs an ancestor-descendant join.

function `join`(L_A, L_D, c_t)

params: L_A – a list of potential ancestors,
L_D – a list of potential descendants,
c_t – a transformation cost,
returns: L – the list of ancestors.

1: $L := [\,]$; $i := 1$; $j := 1$
2: **while** $i < |L_A|$ **do**
3: **while** $j < |L_D|$ **and** $L_D[j]$ is not a descendant of $L_A[i]$ **do**
4: $j := j + 1$
5: $c_{min} := \infty$; $j' := j$
6: **while** $j < |L_D|$ **and** $L_D[j]$ is a descendant of $L_A[i]$ **do**
7: $c_{min} := \min(c_{min}, embcost(L_A[i]) + embcost(L_D[j]) + nodedist(L_A[i], L_D[j]))$
8: $j := j + 1$
9: **if** $c_{min} < \infty$ **then**
10: Append a copy of $L_A[i]$ to L and set its embedding cost to $c_{min} + c_t$
11: $i := i + 1$; $i' := i$
12: **while** $i < |L_A|$ **and** $L_A[i]$ is a descendant of $L_A[i']$ **do**
13: $i := i + 1$
14: **if** $|L_A[i', i]| > 0$ **and** $|L_D[j', j]| > 0$ **then**
15: append the results of `join`$(L_A[i', i], L_D[j', j], c_t)$ to L
16: **return** L

8.4.3 Union and Intersection

The union operator $S_1 \sqcup^{c_t} S_2$ (see Definition 6.9 on page 80) creates the union of the sets S_1 and S_2. If a node appears in only one set, then the cost calculated for the resulting node-cost tuple is the sum of the primary cost of the tuple and the transformation cost c_t. If a node is in both S_1 and S_2, then the minimum of the operand costs is chosen and increased by c_t.

Algorithm 8.3 shows the list-based implementation of the union operator. It makes use of the topological relationships between the nodes to create the result list in linear time with respect to the operand lists. Two cases have to be considered: First, a data-tree node is only represented in one of the lists. In this instance, the entry referring to the node is copied and inserted into the result list. Lines 3–5 capture the case that a node is represented in list L_1 only, whereas Lines 6–8 handle the opposite case. Second, a data-tree node is represented in both lists. Here the entry with the lower embedding cost is chosen (Lines 9–12). In all cases, the embedding cost of the new entry is increased by the cost c_t.

Algorithm 8.3 creates the union of the operand lists.

function `union`(L_1, L_2, c_t)

params: L_1, L_2 – operand lists,
c_t – a transformation cost.
returns: L – the list of entries occurring in L_1 or L_2.

```
1:  L := [ ]; i := 1; j := 1
2:  while i < |L1| or j < |L2| do
3:    if j = |L2| or (i < |L1| and pre(L1[i]) < pre(L2[j])) then
4:      Append a copy of L1[i] to L and increment its embedding cost by ct
5:      i := i + 1
6:    else if i = |L1| or (j < |L2| and pre(L1[i]) > pre(L2[j])) then
7:      Append a copy of L2[j] to L and increment its embedding cost by ct
8:      j := j + 1
9:    else /* pre(L1[i]) = pre(L2[j]) */
10:     cmin := min(embcost(L1[i]), embcost(L2[j])) + ct
11:     Append a copy of L1[i] to L and set its embedding cost to cmin
12:     i := i + 1; j := j + 1
13: return L
```

The intersection operator $S_1 \sqcap^{c_t} S_2$ (see Definition 6.10 on page 81) selects all pairs of tuples from the sets S_1 and S_2 that refer to the same data-tree node. For each pair, it creates a new node-cost tuple consisting of the shared node and a cost, which is the sum of the costs stored in the corresponding tuples plus c_t.

Algorithm 8.4 on the facing page shows the list-based implementation of the intersection

operator. The algorithm iterates through both operand lists simultaneously (Line 2), and selects all pairs of entries that refer to the same data-tree node from the operand lists (Line 3). For each pair, a new entry is created. Its embedding cost is set to the sum of the embedding costs of the operands plus c_t (Line 5). The created entry is then appended to the result list.

Algorithm 8.4 creates the intersection of the operand lists.

function `intersect`(L_1, L_2, c_t)

params: L_1, L_2 – operand lists,
c_t – a transformation cost.
returns: L – the list of entries in both L_1 and L_2.

1: $L := [\,]$; $i := 1$; $j := 1$
2: **while** $i < |L_1|$ **and** $j < |L_2|$ **do**
3: **if** $pre(L_1[i]) = pre(L_1[j])$ **then**
4: $c_{sum} := embcost(L_1[i]) + embcost(L_2[j]) + c_t$
5: Append a copy of $L_1[i]$ to L and set its embedding cost to c_{sum}
6: **if** $pre(L_1[i]) \leq pre(L_1[j])$ **then**
7: $i := i + 1$
8: **if** $pre(L_1[i]) \geq pre(L_1[j])$ **then**
9: $j := j + 1$
10: **return** L

8.5 Complexity Bounds for Operators and Query-Execution Plans

In this section, we summarize the upper bounds for the time and space complexities of the list-based algorithms for operators introduced in this chapter. Based on these findings and on the analysis of query-execution plans done in Section 6.7, we also investigate the upper bounds of time and space consumed during the evaluation of a plan. Because those bounds depend on the chosen query-evaluation method, we analyze the direct method (see Section 8.5.2) and the two schema-driven methods (see Section 8.5.3) separately.

We review the parameters introduced in Section 6.7 and add the letters k, l, m, r, s, s_D, s_P:

- d – maximum number of children of a query selector (query degree),
- k – number of second-level queries,
- l – maximum length of an operand list,
- m – maximum number of nodes in a second-level query,
- n – number of query selectors (query size),
- p – number of permitted permutations for the query,
- r – maximum number of repeated values on a path (recursivity),

s – maximum number of occurrences of a value (selectivity),
s_D – selectivity in a data tree,
s_P – selectivity in a path tree,
v – maximum number of permitted value changes per query selector.

8.5.1 Time and Space Complexities of the Operators

Each operator (except the selection operator) takes two lists and creates a new list that combines the entries of the operand lists. List operators have important properties regarding their time and space complexities:

- The length of the result list created by a join, outerjoin, or intersection operator cannot exceed the length of the longer operand list. The length of the result list created by a union operator is bound by the sum of the lengths of the operand lists.
- The time complexity of each operator is linear with respect to the lengths of the operand lists. For recursive target trees, the execution times of both join operators increase by factor $O(r)$.
- For each operator, the space needed to create a single entry for the result list is $O(1)$. Therefore, the space complexity of each operator is determined only by the size of the result list. For recursive target trees, the join operators need $O(r)$ additional space to maintain position information during recursive calls.

All discussed properties also hold for the operators that handle extended list entries and k-segments. The following table summarizes the time and space complexities of all operators:

	`select`	`join`	`outerjoin`	`union`	`intersect`
time	$O(s)$	$O(r \cdot l)$	$O(r \cdot l)$	$O(l)$	$O(l)$
space	$O(s)$	$O(r + l)$	$O(r + l)$	$O(l)$	$O(l)$

The complexity formula of the selection operator does not take the time needed to access the index into account.

8.5.2 Complexity Bounds for the Direct Evaluation Method

In Section 6.7, we investigated the space complexities of query-execution plans in terms of the number of operators they consisted of. The number of union operators used to represent

permutations or value-changes affects the maximum length of the lists passed during the plan evaluation. Therefore, we first discuss the effect of the list lengths on the time and space complexities of operators, with respect to input parameters like query size and number of permitted value-changes. Next, we relate the operator complexities to the maximum number of operators in a plan in order to get the total time and space complexities of a plan.

The maximum length of a list during the evaluation of a query is determined by the maximum selectivity, i.e., the length of the longest posting used for list initialization. The maximum length of a list increases if value changes are permitted. Each value change of a selector is represented by a selection and a union operator. If a selector has v permitted value changes, then the maximum list length grows by factor $O(v)$. Permutations are similar to value changes: They are added as alternative subplans connected by union operators. If p permutations are defined for the query, and if all of them refer to the same selector, then $O(p)$ union operators are added to the plan and the maximum list length grows by factor $O(p)$. Deletions are the only operations that do not increase the list lengths, because the union operators representing the deletions merge operands taken from the same list. In summary, the upper bound for the length of lists created during the evaluation of an arbitrary query-execution plan is $O(p \cdot v \cdot s)$. To present complexity formulae that abstract from the concrete set of operators used in a plan, we use the complexity bounds of the most time and space-consuming operator: the join operator. We define the abbreviations

$$o_t = O(r \cdot p \cdot v \cdot s) \qquad\qquad o_s = O(r + p \cdot v \cdot s)$$

to denote the worst-case time complexity (o_t) and space complexity (o_s) of any operator, given the maximum list length $O(p \cdot v \cdot s)$. The following table recapitulates the upper bound for the number of plan operators, and summarizes the time and space complexities for the evaluation of a plan:

number of plan operators	evaluation time complexity	evaluation space complexity
$O((n \cdot v + p) \cdot d)$	$O((n \cdot v + p) \cdot d \cdot o_t)$	$O(n \cdot o_s)$

8.5.3 Complexity Bounds for the Schema-Driven Evaluation Methods

The two schema-driven query-evaluation methods proposed in Chapter 7 also use query-execution plans for the creation of second-level queries. However, there are two important differences from the direct evaluation method: First, the operators work on lists with k-segments, which increases both the time and space complexities of the algorithms. Second,

the evaluation of an approXQL query consists of two steps: The creation and the evaluation of second-level queries. In the following, we use subscripts for the letter s to distinguish between the selectivities of the data tree s_D and of the path tree s_P. The time complexity of each operator for extended lists rises by the factor $k \cdot \log k$, which is the time needed to compute cost-sorted segments of size k from the operand lists. The space complexity rises by the segment length k. We define the abbreviations

$$o_t = O(r \cdot p \cdot v \cdot s_P \cdot k \cdot \log k) \qquad o_s = O(r + p \cdot v \cdot s_P \cdot k)$$

to denote the worst-case time complexity (o_t) and space complexity (o_s) of any operator for extended lists, given the maximum list length $O(p \cdot v \cdot s_P \cdot k)$. Because we use the same plans to evaluate a query using the direct method and to create second-level queries, we can adopt our findings about the complexity of the plan evaluation:

number of plan operators	evaluation time complexity	evaluation space complexity
$O((n \cdot v + p) \cdot d)$	$O((n \cdot v + p) \cdot d \cdot o_t)$	$O(n \cdot o_s)$

The algorithm for evaluating second-level queries (Algorithm 7.1 on page 121) visits each node in a second-level query exactly once, and processes lists whose lengths are bound by s_D. Therefore, the evaluation time of a second-level query is bound by

$$O(s_D \cdot m),$$

where m is the number of nodes in a second-level query. The maximum time needed to create and evaluate k second-level queries is

$$O((n \cdot v + p) \cdot d \cdot o_t + k \cdot s_D \cdot m).$$

Note that all parameters of the formula are typically small numbers. This is true even for s_P and s_D, because s_P cannot exceed the number of schema nodes, and s_D is bound by the maximum number of equal paths in the data tree. The results of an experimental efficiency analysis presented in Chapter 10 show that the schema-driven query-evaluation methods are indeed very fast.

8.6 Related Work

The preorder-bound numbering scheme for trees is widely used in retrieval systems that support queries with "skip" operators like "//", or operators that test for indirect inclusion

(see, e.g., [CCB95a, Nav95, Meu00]). Often, the pair of numbers assigned to a node is extended by a third number that stores the depth of the node in the tree. Given two nodes on a path, the depth information allows to verify of whether nodes have parent-child relationships. Amer-Yahia et. al. [ACS02] use the depth difference between two nodes to calculate the degree of relaxation of a query edge. To our knowledge, the integration of cost information into the numbering scheme has not been considered before.

Several researchers proposed efficient algorithms for the implementation of ancestor-descendant joins [Nav95, ZND+01, AKJK+02, CVZ+02]. Navarro [Nav95] demonstrated that ancestor-descendant joins on non-recursive trees can be evaluated in linear time with respect to the lengths of the operand lists. Zhang et. al. [ZND+01] showed that this technique outperforms the evaluation of containment queries using a relational database system. Al-Khalifa et. al. [AKJK+02] proposed two families of joins: tree-merge joins and stack-tree joins. The algorithms for the tree-merge joins are similar to our join algorithm introduced in [Sch01a] and adopted in this thesis: Both the ancestor list and the descendant list are processed simultaneously by exploiting the topology of nodes in the data tree. If the tree is recursive, then parts of the descendant list must be processed several times. The algorithms for the stack-tree joins also process the lists simultaneously, but use stacks to hold the ancestor(s) of the currently processed descendants. Chien et. al. [CVZ+02] proposed an improved algorithm for the evaluation of stack-tree joins that uses B-trees to access the list entries.

Apart from our work published in [Sch01a], the only attempt to integrate a valuation function into an ancestor-descendant join was made by Amer-Yahia et. al. [ACS02] (see our review in Section 6.8). Because the valuation functions of both approaches differ (minimum of costs versus sum of weights), the join algorithms use different techniques to calculate the (partial) scores of the results. Cost-calculating union and intersection operators for the evaluation of XML queries have not been considered by other authors. The algorithms used for those operators are contributions of this work.