Approximate Tree Embedding for Querying XML Data

Torsten Schlieder*
Freie Universität Berlin
schlied@inf.fu-berlin.de

Felix Naumann*
Humboldt-Universität zu Berlin
naumann@dbis.informatik.hu-berlin.de

Abstract

Querying heterogeneous collections of data-centric XML documents requires a combination of database languages and concepts used in information retrieval, in particular similarity search and ranking. In this paper we present an approach to find approximate answers to formal user queries. We reduce the problem of answering queries against XML document collections to the well-known unordered tree inclusion problem. We extend this problem to an optimization problem by applying a cost model to the embeddings. Thereby we are able to determine how close parts of the XML document match a user query. We present an efficient algorithm that finds all approximate matches and ranks them according to their similarity to the query.

1 Introduction

XML has gained much attention in both the information retrieval community and in the field of database research. One reason is that XML offers a uniform and standardized way to represent and exchange both documents in the sense of information retrieval and data exported from databases. On the one end of the spectrum of XML usage are *text-centric* documents with only few, interspersed markups. On the other end of the spectrum are *data-centric* documents that are solely created and interpreted by some application logic. Such documents have a well defined structure and are typically stored in *homogeneous* collections. But XML is also considered as the common format for the *integration* of a wide range of data. One example are federated digital library catalogues that combine the data from several repositories which typically have differing schemata. Another example are data warehouses storing all company-wide XML-formatted documents such as messages and database reports. Such document collections are *data-centric* but do not have a common schema, i.e., they are *heterogeneous*.

To search in *text-centric* XML documents, techniques adopted from information retrieval are appropriate. For querying *data-centric* XML documents in *homogeneous* collections, new query languages have been developed (for an overview, see [FSW99, BC00]). However, for querying *heterogeneous* collections of *data-centric* documents neither information retrieval techniques nor query languages are appropriate. Classic information retrieval models like the vector space model completely ignore the structure of the documents. Even if meaningful element and attribute names are included in the creation of descriptors, the semantics of the nesting is lost. Query languages, on the other hand, cannot cope with documents that are only *similar* to the query. A substructure of a document either matches the query or it does not. The consequence of the binary decision

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

process is that there is no ranking of the results according to the similarity of the query and the document and that often not all desired results are returned. Note, that enriching a query language with *regular path expressions* does not solve the problem of querying collections with partially known schema as we will point out in Section 2.

We believe that for heterogeneous collections of data-centric documents the formalism of database languages combined with the concept of uncertainty used in information retrieval will achieve the best results. In this paper we integrate some aspects of the two concepts and present the XML query language approXQL as a syntactic subset of XQL [RLS98]. Unlike XQL or any other XML query language, results of approXQL are not only all strict matches to the query but also results that have additional inner elements not specified by the query. This method returns satisfying results even when the user does not know the exact structure of the collection. The results of approXQL queries are sorted by their similarity to the query. Exact matches are given first, followed by results with only slight deviations etc. Together with the language we present a query-centric algorithm that efficiently answers approXQL queries.

In Section 2 we show that it is difficult for a user to query a heterogeneous document collection using a traditional XML query language. In Section 3 we introduce the approXQL query language and show how we interpret the queries and the data collection as trees. In Section 4 we review the well-known unordered tree inclusion problem and extend it to the approximate tree embedding optimization problem. In Section 5 we present our algorithm. In Section 6 we survey related work and give an outlook of future improvements and extensions in Section 7.

2 Motivating Example

Assume that we have a collection of documents storing metadata about books. Figure 1 shows a subset of this collection. It contains information about different books the author "Bradley" contributed to.

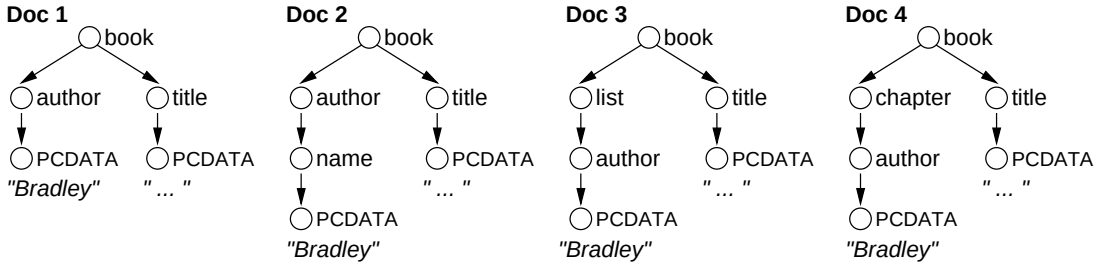


Figure 1: A heterogeneous collection of data-centric documents

Assume further that we need information about all books that have as author "Bradley". We specify an XQL query that exactly models our information need:

```
/book/author/text() = "Bradley"
```

Obviously, the only result of the query is document 1 — despite the fact that at least the documents 2 and 3 exactly match our information need too. Moreover, the document 4 may also be of interest. Fortunately, all XML query languages support *regular path expression*. Even with the simplest kind of regular path expression, the *skip* operator we can partly solve our problem:

```
/book//author//text() = "Bradley"
```

Unfortunately, in order to get the desired results we have to know *that* certain parts of the documents have to be skipped and *where* they are. Coping with structural heterogeneity requires knowledge of the entire data structure and often leads to complex queries. Furthermore, query processors for XML query languages do not measure how many elements have to be skipped in order to match the query. Therefore, no *ranking* according to structural closeness is possible and the results are returned in an arbitrary order.

We assume that the user has only *partial* knowledge of the data structure. The user specifies queries that exactly meet his or her information need. The system retrieves results that exactly fit to the query, but also relaxes the matching conditions to retrieve documents that have additional elements not specified in the query. The results are ranked according to the similarity to the query.

3 Data and Query Modeling

In this section we introduce an elementary data model for XML. We use some normalizations to map elements, attributes and PCDATA to a single node type. We further introduce the approXQL query language.

3.1 Modeling XML Data

We model a collection of XML documents as a labeled tree with a single root. Currently, we ignore ID-references and hyperlinks. To simplify our model, we only use a single node type. Each node d of that type has a label, written as label(d). We use the notation parent(d) to refer to the (unique) parent node of d.

We perform the following normalizations: Elements are mapped to nodes using the element name as label. Each attribute is mapped to two nodes, the first one gets the name as label, the second one gets the value of the attribute as label. PCDATA sections are mapped to nodes by treating the whole text as label. Figure 2 shows an example of the normalization.

(a) Part of a XML data tree

(b) Normalized version

Figure 2: Simplification of a data tree

Note, that we do not consider semantic heterogeneity of attribute and element names. We assume that all elements modeling the same real world domain also carry the same label. This can be achieved by providing a function that maps semantically equal names to a common label.

3.2 The ApproXQL Query Language

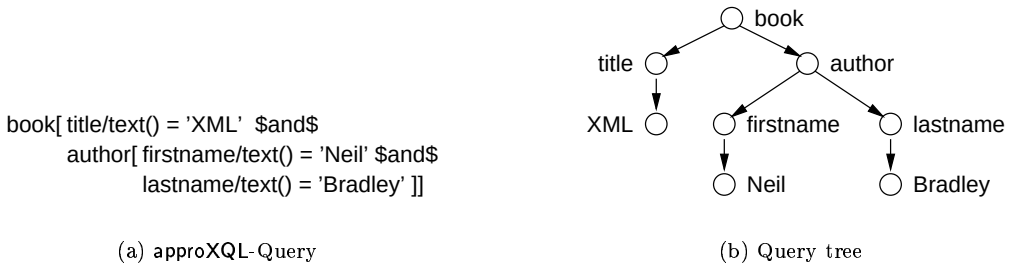
We introduce a simple pattern matching language called approXQL that is a syntactic subset of XQL [RLS98]. Informally, approXQL consists of (1) the path operator "/", (2) the filter operator "`[]`", (3) the `text()` selector, and (4) the operators "`=`" and "`$and$`". The following query requests books that appeared in 1999 and have the author "Bradley":

```
book[year/text() = '1999' $and$ author/text() = "Bradley"]
```

Note, that the user does not need to know how the year 1999 is modeled in the original documents. Due to our simplified data model, attribute values and element contents are both mapped to the leaf nodes of the data tree. Thus, the query will match both cases. We do not require that query paths end with the `text()` selector. The following query matches books that have a foreword and at least one editor:

```
book[foreword $and$ editor]
```

An approXQL query can be interpreted as tree (see Figure 3). Each `$and$` expression is mapped to an inner node of the tree. The children of an `$and$` node are the roots of the paths that are conjunctively connected. We use the notation children(q) to refer to the set of children of q and the notation label(q) to identify the label of q. Note, that different query nodes may have the same label. The result of an approXQL query is always a subtree of the data tree. In contrast to XQL we do not allow queries with boolean results.



(a) approXQL-Query (b) Query tree

Figure 3: Mapping of an approXQL query to a tree.

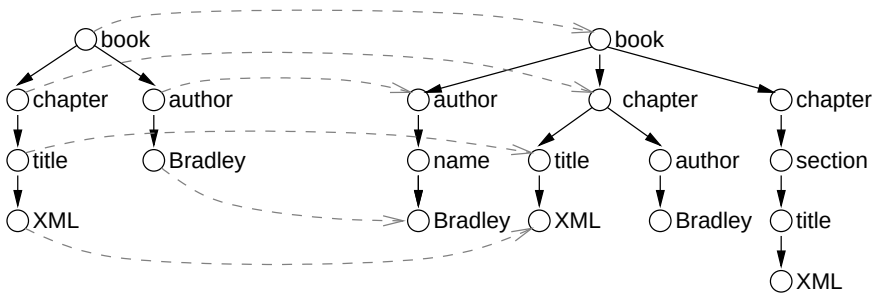
4 The Tree Embedding Problem

We have shown how to interpret both the data and the query as trees. With this interpretation, the problem of answering a query can be mapped to the problem of embedding a query tree in the data tree.

The *exact ordered* tree embedding problem has been extensively studied (see [RR92] for an overview) and is solvable in polynomial time. However, we are not interested in an *ordered* embedding, because ordering in the data may be inconsistent forcing us to ignore it. Even if it were consistent the order might not be known to the user. Also, we are not interested in an *exact* embedding as we will see in the following sections. There, we first review the unordered tree inclusion problem. Then, to answer user queries in a meaningful way, we extend it to an optimization problem.

4.1 The Unordered Tree Inclusion Problem

Our goal is to *approximately* embed the query tree into the data tree such that the labels and the ancestorship of the nodes are preserved. I.e., we allow the data tree to have nodes in between those found in the query tree, but their predecessor–successor relationship must still hold in the tree. Figure 4 shows an allowable embedding. The query tree on the left is not perfectly embedded in the data tree on the right: The name node in the data tree is skipped.



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

Figure 4: Unordered inclusion of a query tree in a data tree.

To find all such embeddings, Kilpeläinen introduced the so called unordered tree inclusion problem [Kil92]. The author proved that even the decision problem, i.e., the problem of deciding whether *one* embedding exists, is NP-complete. We follow [Kil92] in describing the problem.

Definition 1 (Embedding) *A function f from the set Q of query nodes into the set D of data nodes is called an* embedding *if for all $q_i, q_j \in Q$*

1. $f(q_i) = f(q_j) \Leftrightarrow q_i = q_j$,
2. $label(q_i) = label(f(q_i))$,
3. *q_i is the parent of $q_j \Leftrightarrow f(q_i)$ is an ancestor of $f(q_j)$*

We call the data node that is mapped to the query root the *embedding root*. Note the difference between *parent* and *ancestor* in condition 3. Between a node and its parent there exists an edge whereas between a node and its ancestor there exists merely a path.

In [Kil92] the author also studied the problem of locating all subtrees (represented by their root nodes) of the data tree that include the query tree *minimally*. However, in the framework of the author the term "minimal" simply means that only those matching subtrees are retrieved that do not include another matching subtree. In the following subsection we introduce a different interpretation of a minimal embedding.

4.2 The Approximate Tree Embedding Problem

The tree inclusion approach is in itself only of limited value to the user because the solutions are not ranked: The embedding roots are retrieved in arbitrary order independent of how many nodes have to be skipped in order to perfectly embed the query. We do not only want answers that perfectly match the user query. But they should match as well as possible. We solve this problem by measuring the quality of an embedding. We use a cost function over the data nodes that must be skipped to embed the query perfectly.

We use the cost model in two ways: First, among all embeddings that have the same embedding root we only select the cost-minimal embedding. Thus, we avoid situations like that of Figure 4 where several embeddings refer to the same book. Second, we rank the cost-minimal embeddings according to the cost. Embeddings where none or only a small number of nodes are skipped have a low cost and are ranked high. To this end, we assign a deletion cost cost(d) to each data node.

Definition 2 (Embedding cost) *Let f be an embedding of Q in D. Let $f(Q)$ be the image of f and P be the set of data nodes along all paths starting and ending at any two nodes $d_i, d_j \in f(Q)$. Then the* embedding cost *of Q in D is $C := \sum_{d \in P \setminus f(Q)} cost(d)$.*

We define a *match* as an embedding root together with its cost:

Definition 3 (Match) *Let f be an embedding according to Definition 1. Let d be the embedding root. We call the node d together with its embedding cost a* match*: $m = (d, C)$.*

We use the notation d_m and C_m to refer to the components of m. Note, that we also use the term *match* for embeddings of query *subtrees*.

Definition 4 (Minimal match) *Let F be the set of embeddings from a query tree into a data tree and let M be the set of matches belonging to F. We partition M into groups of matches that share the same root node. For each group, the match with the minimal cost is called* minimal match.

The following section describes an algorithm that finds the cost-ranked set of minimal matches to an approXQL query.

5 An Algorithm for the Approximate Tree Embedding Problem

The complexity result of [Kil92] suggests that there is no polynomial algorithm solving the approximate tree embedding problem. We introduce a new algorithm for the approximate tree embedding problem that is still exponential in the worst case. But in typical cases our algorithm only takes sublinear time wrt. the database size. First, we sketch the basic idea of our approach and then we discuss the details of the algorithm.

Our algorithm is based on dynamic programming. Its most important property is the *query-centric* execution. For each query node only those data nodes are fetched that are members of a potential embedding of the query tree in the data tree. The query tree is processed bottom up. For each query node q the embeddings of the query tree rooted at q are combined from the embeddings of the query trees rooted at the child nodes of q. Each embedding is represented by a match. The matches belonging to a certain query node are grouped by their data nodes. From each group only the *minimal* match is selected. The set of minimal matches belonging to the query root are the results of the algorithm. They are sorted by increasing cost and retrieved to the user.

5.1 Constructing the Embeddings

We use a *postorder* numeration of the nodes in the query tree. The main loop of the algorithm visits the query nodes in ascending order (see Algorithm 1, line 1). The postorder ensures that for each child node of the current query node, the minimal matches have already been calculated and stored in the *match set* belonging to that child node. Recall from Section 4 that a minimal match m is a pair consisting of the embedding cost C_m and a data node d_m that is the root of the minimal embedding of a query subtree.

Assume that q_i is the current query node. Now, the algorithm fetches all data nodes that have the same label as q_i (line 3). For each matching data node d, the algorithm tries to embed the query subtree rooted at q_i in the data subtree rooted at d. To construct those embeddings the algorithm builds all combinations of matches where each combination consists of k matches, one from each match set. Each combination represents a potential embedding of the query subtree rooted at q_i in the data subtree rooted at d. To choose the proper combinations out of the set of potential ones, the algorithm checks for each combination S whether all data nodes of the matches in S are descendants of d and whether the matches in S are not blocking. We first formalize the terms *blocking matches* and *proper combination*. Afterwards we explain how the algorithm incrementally constructs the proper combinations.

Algorithm 1 Retrieving the sorted list of minimal matches of a query tree in a data tree.

Input: Q — a list of query nodes sorted in postorder
D — a set of data nodes
Output: The sorted list of minimal matches

```
1:  for all q_i ∈ Q do                                  // Iterate through the query nodes in postorder
2:    M_{q_i} := ∅                                      // The match set belonging to q_i
3:    for all d ∈ D such that label(d) = label(q_i) do
4:      𝕊 := { ∅ }                                      // The set of proper combinations
5:      for all q_j ∈ children(q_i) do
6:        𝕊' := { ∅ }
7:        for all m ∈ M_{q_j} such that d is an ancestor of d_m do
8:          for all S ∈ 𝕊 do                            // At least, ∅ will be selected
9:            if d_m is no ancestor or descendant of any d_{m'} from m' ∈ S then
10:             𝕊' := 𝕊' ∪ { S ∪ { m } }               // Append 𝕊' by a new partial combination
11:           end if
12:         end for
13:       end for
14:       𝕊 := 𝕊'
15:     end for
16:     if 𝕊 ≠ { ∅ } then                               // There is at least one proper combination
17:       M_{q_i} := M_{q_i} ∪ { select_minimal_match(d, 𝕊) }
18:     end if
19:   end for
20: end for
21: output The match set of the query root sorted by increasing cost
```

Definition 5 (Blocking matches) *Let m_i and m_j be two matches with the data nodes d_{m_i} and d_{m_j} respectively. Let $D_i, D_j \subseteq D$ be the node sets of the subtrees rooted at d_{m_i} and d_{m_j} respectively. Two matches m_i, m_j are* blocking *if the subtrees they refer to share common nodes, i.e., $D_i \cap D_j \neq \emptyset$.*

As an example, see Figure 4 on page 5. The query subtrees rooted at the chapter and the author node respectively both have two matches. In particular, they both have one match in the middle part of the data tree. But the data subtree rooted at the chapter node contains the subtree rooted at the author node. Therefore, those matches are blocking.

The blocking of two subtrees can be determined efficiently: Two subtrees are blocking if their root nodes are identical or if the root node of one subtree is an ancestor of the other (line 9).

Definition 6 (Proper combination) *Let q_i be a query node and d be a data node to which q_i is mapped. Let M_{q_j} be the match set belonging to $q_j \in children(q_i)$. Each tuple S of the cartesian product $X_{q_j \in children(q_i)} M_{q_j}$ is called a* proper combination *if (1) the data node d_m of each match $m \in S$ is a descendant of d and (2) if each pair of matches $m_i, m_j \in S$ is non-blocking.*

Our algorithm incrementally constructs the proper combinations for a query node q_i and a data node d. First, the set of proper combinations is empty (line 4). The temporary set of combinations $\mathbb{S}'$ just collects all matches attached to the first child node of q_i (lines 7—13). Then, $\mathbb{S}$ is set to $\mathbb{S}'$ (line 14). The second loop combines all matches of the second child with the matches of the first child now stored in $\mathbb{S}$. That is, combinations of length two are constructed. The loop beginning at line 5 finishes if all k child nodes of q_i are processed, i.e., if proper combinations of length k are constructed. Each proper combination S represents an embedding of the query subtree rooted at q_i in the data subtree rooted at d.

We are only interested in the *minimal* embedding for q_i and d. Therefore, the embedding cost of each combination S has to be calculated. In Section 5.3 we introduce an algorithm that efficiently calculates the embedding costs in order to select the minimal match. In the following subsection we show how the ancestor-descendant relationship of two data nodes can be tested efficiently. This crucial test is both needed in Algorithm 1 and for the calculation of the embedding cost.

5.2 Testing the Ancestor-Descendant Relationship

In a tree, the test whether a node is an ancestor of another node can be done in constant time after a linear time preprocessing of the tree [Hav97]. This method uses a *preorder* numeration of the nodes. If a node d_j is a descendant of a node d_i then $\text{number}(d_i) < \text{number}(d_j)$. If the comparison is true, then d_j is either a descendant of d_i or d_j is in a subtree right of d_i. The latter case can be excluded if the number of the right-most leaf node of d_i is known. We explicitly store this $\log(|D|)$ number for each node d_i and access it with the function $\text{bound}(d_i)$. Now the test whether a node d_i is an ancestor of a node d_j can be performed as follows:

$$d_i \text{ is an ancestor of } d_j \iff \text{number}(d_i) < \text{number}(d_j) \textbf{ and } \text{bound}(d_i) \geq \text{number}(d_j).$$

5.3 Calculating the Embedding Cost

To calculate the embedding cost for a given data node d and a proper combination S two steps are necessary. First, we sum up the cost attached to the matches in S. Second, for each data node d_m belonging to a match $m \in S$ we add the cost of the path between d and d_m. This is performed by traversing the parent links starting at d_m and ending at d. During traversal we add the delete cost $\text{cost}(d_i)$ of each visited node d_i. Note, that the paths may share common nodes — but for each node, the delete cost is included only once.

The preorder numeration together with the right-most leaf node numbers can be used to efficiently calculate the embedding cost. In Algorithm 1 the embedding cost of a query node q_i and a data node d is calculated using a set of proper combinations $\mathbb{S}$ belonging to the child nodes of q_i (line 17). In the following we assume that each $S \in \mathbb{S}$ is a list sorted ascendingly by the preorder node numbers. The operator $\text{shift}(S)$ retrieves the first component (i.e., match) of S and deletes this component from S.

S has three important properties that follow from its construction:

1. d is an ancestor of all d_m, $m \in S$,
2. no node d_{m_i} is an ancestor or descendant of another node d_{m_j}, and therefore
3. if $\text{number}(d_{m_i}) < \text{number}(d_{m_j})$ then d_{m_j} is in a subtree to the right of d_{m_i}.

These properties lead to a simple algorithm for calculating the embedding cost (see Procedure 1). It calculates the embedding cost starting at the left-most data node d_m being part of a match $m \in S$. Because of the preorder numeration this is the node with the smallest number among all data nodes belonging to matches in S. The algorithm traverses the path starting at d_m to the root node of the embedding d_r setting d_i to each inspected node (line 2). The delete cost $\text{cost}(d_i)$ is summed up while traversing the path (line 3). Whenever another matching data node d_m is a descendant of d_i then the procedure `calculate_cost` is called recursively using d_i as root and d_m as the new left-most node (line 5). After the path costs for all matching nodes in the subtree rooted at d_i have been added the algorithm proceeds with the parent of d_i (line 8). Note, that for each $m \in S$ the path between d_m and d is traversed only once. Therefore, the runtime complexity

Procedure 1 Calculating the embedding cost.

Procedure `calculate_cost`(d_r, d_i, C, S)
Input: d_r — the root node of the subtree for which the cost is calculated
d_i — the data node currently inspected
C — the embedding cost calculated so far
S — a proper combination (in-out parameter)
Return: The embedding cost

1: $m :=$ shift(S)
2: **while** number$(d_i) \geq$ number(d_r) **and** $(d_j =$ **nil or** bound$(d_i) <$ number$(d_m))$ **do**
3: $C := C +$ cost(d_i)
4: **while** $d_m \neq$ **nil and** bound$(d_i) \geq$ number(d_m) **do**
5: $C := C +$ `calculate_cost`$(d_i, d_m, C, S) -$ cost(d_i)
6: $m :=$ shift(S)
7: **end while**
8: $d_i :=$ parent(d_i)
9: **end while**
10: **return** C

is bound by $O(k \cdot h)$ where k is the maximal number of children of any query node and h is the height of the data tree.

The only step left is an algorithm that selects the minimal embedding and retrieves the accompanying minimal match to the Algorithm 1. This is performed by the simple Procedure 2 that uses Procedure 1 as subroutine.

Procedure 2 Selecting the minimal match.

Procedure `select_minimal_match`$(d, \mathbb{S})$
Input: d — a data node that is the root of the subtree embeddings
$\mathbb{S}$ — a set of proper combinations
Return: The minimal match

1: $C := \infty$
2: **for** $S \in \mathbb{S}$ **do**
3: $m :=$ shift(S)
4: $C := C_m +$ min$(C,$ `calculate_cost`$(d, d_m, 0, S))$
5: **end for**
6: **return** (d, C)

5.4 Runtime Complexity

The tree inclusion problem which is contained in our approximate tree embedding problem is NP-complete. However, real world instances often have favorable properties. Our algorithm is especially designed for such instances and in fact solves them in polynomial or even sublinear time, yet its runtime complexity remains exponential. First, we analyze exactly the worst-case complexity and then we discuss why the algorithm is nevertheless fast for queries and data used in practice.

The outer loop of Algorithm 1 iterates over the $|Q|$ query nodes (line 1). For each query node q_i there are at most $s \leq |D|$ data nodes that have the same label as q_i (line 3). We assume that fetching those matches needs I cycles, e.g., $I = \log|D|$. The loop at line 5 iterates over all k children of q_i. Assume that we are at jth child $(1 \leq j \leq k)$. There are at most s^{j-1} (partial)

combinations in $\mathbb{S}$. Each combination $S \in \mathbb{S}$ consists of exactly $j-1$ matches. We have s matches for the current child. This leads to $s^{j-1} \cdot (j-1) \cdot s = (j-1) \cdot s^j$ comparisons. Note, that the set of combinations $\mathbb{S}$ may grow exponentially. Thus, at the end of the iteration over all child nodes of q_i, we needed at most $\sum_{j=2}^{k}(j-1) \cdot s^j = O(k^2 \cdot s^k)$ loops. The calculation of the embedding cost of one proper combination (see Procedure 1) can be done in $O(k \cdot h)$ where h is the height of the data tree. Because there are at most s^k proper combinations, the selection of the minimal match takes at most $O(s^k \cdot k \cdot h)$ cycles (Procedure 2). Therefore, the overall runtime complexity is

$$O(|Q| \cdot I \cdot s \cdot (k^2 \cdot s^k + s^k \cdot k \cdot h)) \quad = \quad O(|Q| \cdot I \cdot s^{k+1} \cdot k \cdot (k+h)).$$

One should not be scared off by the worst case runtime complexity of the algorithm. First, no factor of the formula depends *directly* on the size of the data. The number of data nodes influence only the time for the index lookup I, the selectivity s, and the height of the data tree h. Second, the only factor that rises exponentially is s^k. This factor represents the number of proper combinations belonging to a query node and a data node. It rises exponentially only in pathological cases, e.g., if all query nodes and data nodes carry the same label. For real data, this factor is typically *decreasing* during query processing. If the tree structure is non-recursive (i.e., no node has an ancestor with the same label) then the number of proper combinations is bound by $O(K^k)$ where K is the maximal number of children of a data node. If both k and K are bound by a constant then K^k is just a constant value. First experiments with our prototypical implementation show that the above arguments are true in practice. But further experiments have to be done to confirm our claim.

6 Related Work

Our work has four related areas: query languages for semistructured data, structural queries in information retrieval, structure matching in graphs, and distance metrics for trees.

Many query languages for semistructured data and XML in particular have been developed during the last years [AQM+97, RLS98, DFF+98, CJS99, Moe00]. All languages support operators that allow to skip certain parts of the data. However, as already mentioned, they require that the user knows which parts have to be skipped. Furthermore, they do not consider how many nodes have to be left out in order to match the query. Thus, no ranking depending on structural similarity is supported.

There are also many proposals of query languages in the field of information retrieval. Here, the endeavor is to integrate content and structure in the query answering process. A good overview of existing methods can be found in [NBY96]. Again, to the best of out knowledge no approach measures the structural similarity between the query and the documents.

The problem of tree and graph embedding has also been studied in the theory of graphs and in formal query answering models. The work most closely related to ours is [Kil92] that introduced the tree inclusion problem. This approach was also taken up and modified in [Meu98]. Both approaches do not measure the closeness of the query tree and the data tree. Other tree and graph matching approaches, e.g., [BF99, NS00] also do not quantify the similarity between the query and the data.

Several measures for the similarity of trees have been developed [Tai79, JWZ94]. In the *ordered* case they are computable in polynomial time. But as we argued in this paper, ordered mappings are not useful for querying XML data. The problem of finding the minimal edit or alignment distance between unordered trees is MAX SNP-hard [AG97]. The complexity results suggest that both measures are not useful for querying trees. Even algorithms for restricted forms of the edit distance [Zha93, ZWS95] are bound by $O(|Q| \cdot |D| \cdot K)$ where K depends on the maximal degree of the nodes. We believe that algorithms that touch every data node are not applicable for large databases.

7 Outlook

For future research we conceive two directions—improvement of the algorithm power and execution time and improvement of the usefulness for the user.

Goldman and Widom introduced *strong DataGuides* as a schema for semistructured data [GW97]. Our algorithm can make use of such schemata to greatly accelerate the search for embeddings in the data tree. Further optimization techniques make use of certain properties of the query and the data. Also, we plan to relax the tree assumption and allow directed acyclic and even cyclic data structures.

The current version of our algorithm compensates under-specification of the user query: Missing nodes in the user query are skipped in the data. An over-specified user query has nodes that do not appear in the data. We plan to compensate by skipping nodes in the query in the same way as we do in the data. Further, we plan to investigate methods for learning the delete costs. Currently each node has the same delete cost regardless of the importance of the node.

References

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

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

[BC00] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1), March 2000.

[BF99] A. Bergholz and J. C. Freytag. Querying semistructured data based on schema matching. In *Proceedings of the International Workshop on Database Programming Languages (DBPL)*, September 1999.

[CJS99] S. Cluet, S. Jacqmin, and J. Siméon. The new YATL: Desing and specifications. Technical report, INRIA, 1999.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. W3C Note, August 1998. http://www.w3.org/TR/NOTE-xml-ql.

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

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured data. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, pages 436–445, Athens, Greece, August 1997.

[Hav97] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[JWZ94] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Combinatorial Pattern Matching*, pages 75–86, June 1994.

[Kil92] Pekka Kilpeläinen. *Tree Matching Problemas with Applications to Structured Text Databases.* Report A-1992-6, University of Helsinki, Finnland, November 1992.

[Meu98] H. Meuss. Indexed tree matching with complete answer representations. Workshop on Principles of Digital Document Processing (PODDP), 1998.

[Moe00] G. Moerkotte. YAXQL: A powerful and web-aware query language supporting query reuse. Technical report, University of Mannheim, January 2000.

[NBY96] G. Navarro and R. Baeza-Yates. Integrating content and structure in text retrieval. *SIGMOD Record*, 25(1):67–79, March 1996.

[NS00] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the 19th Symposium on Principles of Database Systems (PODS)*, May 2000.

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

[RR92] R. Ramesh and L.V. Ramakrishnan. Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295–316, April 1992.

[Tai79] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[Zha93] K. Zhang. A new editing based distance between unordered labeled trees. In *Combinatorial Pattern Matching*, pages 254–265, June 1993.

[ZWS95] K. Zhang, J.T.L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Combinatorial Pattern Matching*, pages 395–407, July 1995.