Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 216 (2008) 93–109
www.elsevier.com/locate/entcs

Phil: A Lazy Implementation of a Language for Approximate Filtering of XML Documents

M. Baggi and D. Ballis

Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy.
Email: {baggi,demis}@dimi.uniud.it.

Abstract

In this paper, we introduce a system, written in Haskell, for filtering information from XML data. Essentially, the system implements a simple declarative language which allows one to extract relevant data as well as to exclude useless and misleading contents from an XML document by matching patterns against XML documents.
The matching mechanism employes a cost-based pattern transformation algorithm which searches for patterns in an approximate way (i.e. modulo renaming, insertion, and deletion of XML items) and ranks the results w.r.t. their cost. In order to improve efficiency, the implementation uses sophisticated indexing techniques and exploits laziness to automatically avoid the construction of unnecessary data structures.
We analyzed both the expressiveness of our filtering language and the performance of the system using the well known XMark benchmark suite.

Keywords: XML filtering and query languages, approximate pattern matching, tree embedding problem.

1 Introduction

The adoption of XML[13] as a widely accepted standard for data representation and exchange has led to a rapid growth in the amount of XML data available over the internet. Nowadays, large-scale XML repositories are constantly browsed, queried and modified by internet users, who typically retrieve a lot of information which is not always possible to absorb in a pleasant and/or understandable fashion. In order to tame the inherent complexity of such a massive amount of data, a lot of research effort has been invested by computer scientists in the last years giving rise to a proliferation of decision-support systems to manage and explore XML repositories.

Arguably, a growing attention has been dedicated to query and filtering languages as means to efficiently extract all and only the relevant information from huge data collections. As a matter of fact, information frequently appears obscure or difficult to interpret; moreover, most of the time, just a small percentage of the whole amount of the data received is considered interesting by the user. Therefore, query and filtering systems represent a valid way to obtain those contents which

doi:10.1016/j.entcs.2008.06.036

best fit user's needs. The World Wide Web Consortium has defined XQuery[15] and XPath[14] as standard languages to consult and filter information in XML documents, nonetheless a plethora of alternative and worthwhile proposals have been developed independently, e.g. [4,8,6]. Basically, they all work by *exactly* matching a given pattern (or path expression) representing the information to be searched for against an XML document. Hence, recognized pattern instances are delivered to the user.

Although the languages mentioned above are very advantageous in many applications, they may be of limited use when dealing with data filtering in a pure information retrieval context, since (i) they require the user to be aware of the complete XML document structure, (ii) results that are not fully matched are not delivered, (iii) there is no result ranking. Therefore, in this context, a more flexible matching mechanism which can manage the lack as well as the vagueness of the information is necessary. Such an *approximate* behaviour is not typically implemented in the standard query languages, and actually only few works address this issue.

For instance, the PIX[10] system is a phrase matching system tailored to XML for searching a given phrase in an XML document. It implements a rough approximate matching method which basically allows to ignore some tags included in the document, while no deletion and renaming of XML items are permitted.

A more flexible approach is followed in ApproxXQL[12,11], which is an approximate query language which provides a more sophisticated approximate matching mechanism. It is based on a cost-based query transformation algorithm which allows to rename, insert and delete XML items in order to find the best match between a pattern and a given XML document. However, this language is still rather simple and does not offer the full expressive power of modern query languages.

Our contribution

The original contribution of this paper is twofold.

(i) We present a novel declarative language for approximate filtering of XML documents, which allows the user to easily select the desired information (*positive filtering*) as well as to remove noisy, spurious data (*negative filtering*) from a given XML document. Our language is easy to use and thus can be employed even by those users who are typically not used to express themselves using formal methodologies, since no special expertise is required. Basically, in our approach, XML documents and filtering queries are encoded as tree-shaped terms of a suitable term algebra, then an approximate tree embedding algorithm is employed to execute filtering queries on XML documents to recognize the information that the user wants to select or to strike out. Our approach is inspired by ApproXQL and extends it in several ways.

- ApproXQL allows to define only ground patterns, while our language provides pattern variables, which can be used to extract parts of the document on which we can perform further tests.
- We add regular expressions and built-in functions to model conditional fil-

tering rules with the aim of refining the approximate search engine.
- Nested filtering queries are allowed, while ApproXQL manages only flat queries.
- ApproXQL does not support negative filtering, while our language does. This feature allows to introduce the expressive power of negation in the language. As a matter of fact, within our framework, we can easily formulate rules to answer queries of the form: "Which people don't have a homepage?"

Besides, the approach followed in this paper improves our previous filtering framework [2] which formalizes an exact tree embedding algorithm for filtering XML documents.

(ii) The filtering language has been implemented in the prototypical system PHIL using a lazy functional language (Haskell). Through a thorough experimental evaluation, we will show the convenience of such a lazy implementation and the usefulness of the undertaken approach. In particular, we will illustrate how laziness automatically avoids the construction of unneeded data structures and allows to directly obtain a simple and efficient implementation of the filtering language.

Plan of the paper.

The rest of the paper is structured as follows. In Section 2, we formalize our filtering language, while Section 3 illustrates how the filtering problem can be reduced to a tree matching problem. In Section 4, we outline our approximate pattern matching algorithm. Section 5 provides some experiments and a qualitative as well as quantitative evaluation of the implementation of our methodology with a particular focus on the benefits offered by lazy functional languages in this application domain. Finally, Section 6 concludes.

2 The Filtering Language

The filtering language we describe is a declarative, pattern-based language in which we can specify filtering rules as (possibly conditional) patterns. A filtering rule matches an XML document if the pattern is somehow "embedded" into the XML document and fulfills the desired relationships and conditions. Basically, a filtering rule can be formalized by means of the following syntax:

```
{count} <filterop> <pat> in <XML doc> where <cond> (<mode>)
```

which informally says that

(i) a pattern `pat` is searched in a document `XML doc`;

(ii) only detected instances of `pat` which satisfy the given condition `cond` are either extracted (*positive filtering*) or removed (*negative filtering*) from `XML doc` according to the value of the filtering mode `mode`. A filtering mode is a label belonging to the set {`P`,`N`}. Positive filtering rules are identified by means of the filtering mode `P`, while the negative one are denoted by `N`. Whenever a filtering rule does not specify a filtering mode, it has to be considered a positive

filtering rule.

(iii) `count` is an optional operator which allows to count the number of pattern instances detected in the given XML document.

Moreover, several filtering operators `filterop` have been formulated to support approximate as well as exact matching mechanisms. Finally, note that, when no condition is specified, the `where` part of a filtering rule can be omitted.

In the remainder of this section, we present the syntax of each component of a filtering rule providing a brief explanation of the basic constructs of the language.

Filtering operators.

We provide four filtering operators which can model both exact and approximative filtering w.r.t. a universal as well as existential semantics.

- `filterOneBest` is an operator that allows one to search for the best approximate match between a given pattern and an XML document. Approximate matching has to be intended modulo renaming, insertion and deletion of pattern items. More precisely, when no exact match is found, either some tag items of the filtering pattern may be renamed or new elements may be inserted/removed in order to find an approximate match. Any insertion/deletion/renaming operation has a fixed cost. `filterOneBest` returns the match with lower cost.

  Informally speaking, given a pattern, we try to generate a result for every position in the document where there is a tag that matches the pattern root. `filterOneBest` then selects the position referring to the document subpart that better matches the pattern. When there is more than one best approximate match, the operator will only deliver the first one it discovered.
- Since there might be several matches with the same cost deriving from the application of distinct sequences of insertion, deletion and renaming operations, the `filterAllBest` operator returns all the best approximate matches found, i.e. all the matches of minimum cost.
- `filterOneExact` is an operator that exactly matches a specified pattern against an XML document. In case that more than one exact match is detected, this operator will only deliver the first one it discovered.

  "Exactly" means that the labels and the structure of the pattern are preserved and precisely recognized inside the XML document. In other words, no renaming, insertion and deletion of pattern items are allowed to "adapt" the pattern to the given document.
- `filterAllExact` returns all the exact matches found.

Patterns.

Patterns of filtering rules are used to describe the information we want to detect inside a given XML document. A pattern is built by composing the following syntactical elements.

- *Variables* (we assume to have a countable infinite set of variables {`X,Y,...`}).

- *Text selectors*, that is strings of plain text surrounded by single quotes (e.g. `'Dear friend'`). Text selectors will be matched against the textual part of the XML document.
- *Tag selectors* represent XML tags and are denoted by strings of characters (e.g. `author`, `book`,...). Tag selectors can be followed by the *occurrence* operator `[i]`, where `i`$\in \mathbb{N} \cup \{\text{last}\}$. Given a sequence of terms (i.e., XML documents) all rooted by a tag `t`, `t[i]` selects the `i`-th term. The keyword `last` is used to select the last element of the sequence (e.g. `book[1]`, `book[last]`).

  Moreover, tag selectors can be used together with the *synonymity* operator "`$`", which enables the flexible matching of tag selectors. More precisely, given a tag selector `t`, `$t` allows to match `t` against any synonym of `t` which has been defined by the user. Synonyms of tag `t` can be seen as alternative items w.r.t. `t`, that can be employed in an approximate search.
- The containment operator is represented by brackets "`()`" and it is used in combination with tag selectors and boolean operators to define boolean-connected structured patterns. Given a tag selector `t` and $\texttt{pat}_1, \ldots, \texttt{pat}_n$ patterns, the following syntactical expressions are legal patterns:
  - $\texttt{t}(\texttt{pat}_1, \ldots, \texttt{pat}_n)$. The comma "`,`" separator represents the logical conjunctive operator and allows to build conjunctions of patterns. For instance, the pattern `book(title(X),author(Y))` searches for all the books containing both a title `X` *and* an author `Y`.
  - $\texttt{t}(\texttt{pat}_1 | \ldots | \texttt{pat}_n)$. The "`|`" separator allows to model boolean disjunctions of patterns. For example, `book(title(X) | author(Y))` selects all the `book` instances containing a title `X` *or* an author `Y`.
  - $\texttt{t}(\texttt{pat}_1 ? \ldots ? \texttt{pat}_n)$. The separator "`?`" formalizes the boolean *xor* operator. One may use this operator to obtain the evidence of the existence of exactly one of the patterns in the list.

  Operators "`,`" , "`|`" and "`?`" are called *inner boolean operators*. Note that brackets "`()`" are also used to specify precedence in boolean-connected patterns as shown in the following example:

```
pubs(report(author(X),year('2007'))|article(author(Y),year('2007')))
```

- Several patterns can be connected together at the root level by means of the outer boolean operators ("`and`", "`or`", "`xor`"). Given patterns $\texttt{pat}_1, \ldots, \texttt{pat}_n$, the syntactical expression $op(\texttt{pat}_1, \ldots, \texttt{pat}_n)$, where $op \in \{\texttt{and}, \texttt{or}, \texttt{xor}\}$ is still a legal pattern.

  Although, outer and inner boolean operators behave very similarly, there is a subtle difference between them. Roughly speaking, when an inner boolean operator is used, it always refers to a parent node explicitly. For example, in the pattern `h(f(X),g(Y))`, the parent node of operator "`,`" is the tag selector `h`, so the patterns `f(X)` and `g(Y)` are somehow connected to the parent node `h`. When an outer operator is used, the parent node is not specified, and the boolean-connected patterns are executed independently.

XML documents and nested filtering rules.

Filtering rules work on XML documents. There are three ways to supply an XML document to a filtering rule:

- directly giving the XML code. For instance, `filterOneBest a(X) in <a>b</a>`.
- giving the name of a file containing the XML data. In this case, the keyword `file` must precede the file name to be loaded. For instance,

  ```
  filterOneExact a(X) in file 'test.xml'.
  ```

- The execution of a filtering rule generates an XML document. Thus, the outcome of a filtering rule may be employed to feed another filtering rule. Or, equivalently, the result of an inner rule becomes the source document for an outer rule. Our language supports nested filtering rules with an arbitrary level of nesting. As an example, consider

  ```
  filterOneBest a(X) in
   (filterAllBest b(a(X),c()) in file 'test.xml' where X match [fg]*)
  ```

Conditions.

The condition is an optional part of the filtering rule, which can be employed to further refine the search of a given pattern inside an XML document. Formally, a condition is a sequence $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_n$, where each $\mathtt{c}_i$ can be

- a membership test of the form `X match RegExp`, where `X` is a variable occurring in the pattern and `RegExp` is a regular expression [1]. If the variable X is bound to a complex XML subtree (not just a textual node), we build up a string `s` concatenating the labels of all the textual nodes in the subtree, traversing it from left to right, and we subsequently check whether `s` belongs to the language denoted by the considered regular expression `RegExp`.
- an equation `s=t`, where `s` and `t` are terms built over a set of primitive operators and the set of variables occurring in the pattern. Note that terms may be non linear, that is, they may contain multiple occurrences of the same variable. Our language supports a number of built-in operators to deal with strings and numbers (arithmetic operators, string concatenation, equality over numbers and strings, *etc.*).

Counting the results.

By executing a filtering rule on an XML document `doc`, we generate a new XML document containing one or more instances of a given pattern that are embedded into `doc`. However, we might be interested only in the number of embeddings found (e.g. we want to know the number of books written by an author). To model this feature, our language is equipped with the *counting* operator `count` which takes a filtering rule f and a maximum cost c as arguments. The result of applying the

[1] Regular languages are represented by means of the usual Unix-like regular expressions syntax [9].

`count` operator to f and c is the number of embeddings found by executing f whose cost does not exceed the value c.

Some examples.

The following examples formalize, within our framework, three queries (namely, query **Q3**, query **Q5**, and query **Q17**) of the XMark benchmark set [5], which is typically used to evaluate XML filtering and query languages. More precisely,

Q3: Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.

```
filterAllExact id(Z) in
 (filterAllBest $site($open_auctions(open_auction(id(Z),
                                     bidder[1](increase(X)),
                                     bidder[last](increase(Y)))))
      in file 'auction.xml' where 2*X <= Y (P))
```

Q5: How many sold items cost more than 40?

```
count 0 (filterAllExact price(X) in
   (filterAllBest site($closed_auctions(closed_auction(price(X))))
               in file 'auction.xml' where X >= 40 (P)))
```

Q17: Which people don't have a homepage?

```
filterAllExact site(people(person(name(X)))) in (
   filterAllBest site(people(person(name(X),homepage(Y))))
   in file 'auction.xml'(N))
```

Note that we introduced some occurrences of the "`$`" operator to explicity allow the flexible matching of some tag selectors.

3 Filtering is a Tree Embedding Problem

Filtering can be treated as a matching problem over trees. In fact, filtering rule patterns and XML document can be straightforwardly encoded into tree-shaped terms of a suitable term algebra. Note that XML tag attributes can be encoded into common tagged elements and hence translated in the same way (for further information, see [1]).

On the one hand, a pattern can be interpreted as a tree in the following way:

- each variable and text selector is mapped to a leaf node;
- each tag selector and boolean operator is mapped to an inner node;
- the containment operator is interpreted as the standard tree parent-child relation.

The tree representing the pattern is called *pattern tree*. Figure 1(a) illustrates the tree encoding of the pattern `h(f('a',X)|g(Y?m('b')))`.

On the other hand, XML documents are provided with a tree-like structure in which plain text elements are mapped to leaf nodes, while tag elements define the

inner structure of the tree. Note that XML tag attributes can be considered as common tagged elements, and hence translated in the same way. Precisely, the following piece of XML `<tag att1="val1" ... attn="valn"> ... </tag>` can be first translated into `<tag><att1>val1</att1>... <attn>valn</attn> ...</tag>`, and next encoded into a tree as described above. The tree representing the XML document is also called *data tree*.

By interpreting patterns and documents as trees, executing a filtering rule boils down to finding one (or all) the matches (i.e., embeddings) of a given pattern tree into a data tree. The recognized subtrees are then either selected or removed from the data tree according to the filtering mode of the rule. Therefore, our filtering mechanism is inspired by and slightly modifies the *unordered path inclusion problem*[7]. Roughly speaking, the unordered path inclusion of a tree T_1 in a tree T_2 is defined as an *injective* function from T_1 to T_2 that preserves labels of the nodes and parent-child relationships (i.e. the structure), but not the order of the siblings. Equivalently, it can be considered as a particular instance of the Kruskal's embedding relation[3]. We think that ignoring the order of siblings is favorable or even necessary for filtering XML data, because the ordering of the XML items may not be known to the user.

Following [12], our methodology discards the injectivity property, which is required in the path inclusion problem. Although this can imply a possible loss of precision of the computed results, the efficiency of the matching method is greatly improved, since the computed non-injective embeddings encompasses several injective embeddings at the same time. Moreover, while the unordered path inclusion problem typically searches for *exact* answers (in a sense that the labels and the structure of the pattern tree are precisely embedded in the data tree), our goal is to find a match even when no exact instances of the pattern tree can be recognized inside the data tree.

To find such *approximate* results, we use pattern transformations, which minimally modify the original pattern tree and adapt it to the data tree with the aim of finding the best match which now might be not precise.

A pattern transformation consists of a sequence of basic transformations. Each basic transformation has a cost which is represented by a natural number. The

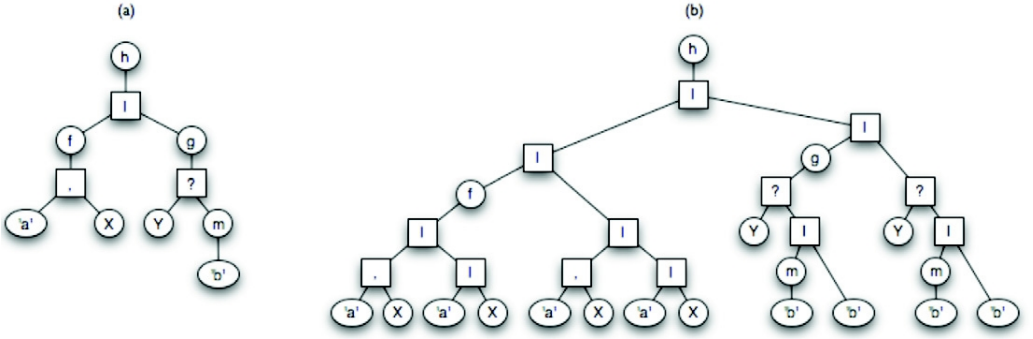


Figure 1. Tree encodings of a filtering rule pattern.

total cost of a sequence of basic transformations is assigned to the matching result of the transformed pattern tree against the data tree and used to rank the result by increasing cost. We consider the following three types of basic pattern transformations.

Renaming. A label l of a pattern tree inner node can be renamed with a new label l' provided that l' is a *synonym* of l. The synonymity relation might be explicitly provided by the user or automatically computed by querying an ontology. Renaming is enabled only for tag selectors to which the synonymity operator "`$`" is applied.

Deletion. A pattern tree node (corresponding to either a tag selector or a text selector) can be deleted, whenever it is not the pattern tree root.

Insertion. A new tree node (corresponding to a tag selector) can be inserted into the pattern tree. However, it is not allowed to add a new root or new leaf nodes. In particular, leaf nodes cannot be inserted, because they represent user-dependent data which cannot be automatically inferred by the algorithm.

The costs associated with these basic pattern transformations will be denoted respectively as renaming/deletion/insertion cost. In the next section, we describe a pattern-transformation algorithm which implements the strategy mentioned above.

4 An Approximate Tree Matching Algorithm

We start describing a basic procedure for approximate tree matching for ground patterns. Next, we will add all the other components of a filtering rule (variables, conditions,...). The core algorithm is a slightly modified version of the one proposed by Schlieder in [11] which not only finds a single best match, but also allows to find all the best matches of a pattern tree w.r.t. a data tree.

From a theoretical point of view, to evaluate a pattern tree $\mathcal{P}$ against a data tree $\mathcal{D}$, we can follow these steps:

1 Derive from $\mathcal{P}$, every pattern tree $\mathcal{P}'$ which is obtained by applying a sequence of basic transformations to $\mathcal{P}$ (i.e. we compute the *pattern transformation closure* w.r.t. the basic transformations) and compute the corresponding total cost $c_{\mathcal{P}'}$.

2 Find all the exact matches of $\mathcal{P}'$ against $\mathcal{D}$, for each $\mathcal{P}'$.

3 Group the matches found into embedding sets, where an embedding set is a set of matches which refer to the same subtree of the data tree.

4 From each embedding set, choose either one match or all the matches with lower cost $c_{\mathcal{P}'}$ according to the filtering operator applied.

5 Rank the selected matches according to their costs.

Obviously, the brute-force generation of the pattern transformation closure is not feasible, since it would lead to an infinite set of transformed patterns. Nonetheless, in the following, we will show a possible way to solve the problem, which exploits smart representations of data and pattern trees. Basically, all possible node dele-

tions will be encoded in a single pattern tree, while renamings and insertions will be encoded into an index modeling the data tree.

4.1 Data Tree Encoding

As shown in Section 3, an XML document can be represented by means of a data tree in which leaf nodes represent plain text items and inner nodes represent XML tags. In the following, given a node u of a data tree $\mathcal{D}$ the *position* (or *preorder number*) of u in $\mathcal{D}$ is a natural number $n \geq 1$ assigned to u by a preorder traversal of $\mathcal{D}$. The position of the root node is 1.

In order to construct an approximate match of a pattern tree w.r.t. a data tree, some nodes may need to be inserted into the pattern tree or simply renamed. To avoid the explicit insertions of nodes into a pattern tree, we use a special encoding of the data tree which measures the insertion distance between two nodes in a data tree. Formally, given two nodes u and v of a data tree $\mathcal{D}$ such that u is an ancestor of v, the *insertion distance* between u and v is the sum of the insertion costs of all nodes along the path from u to v (excluding u and v). Moreover, information regarding label renaming of pattern tree nodes is formalized by providing an extensional representation of the synonymity relation (i.e. any node label is decorated with the list of its possible synonyms) along with the associated renaming cost.

The encoding is based on an indexing technique that is inspired by the *partial index* data structure which has been originally introduced in [12] and then successfully employed in the language ApproXQL[11]. The data structure is as follows.

Given a data tree $\mathcal{D}$, for each node u appearing in $\mathcal{D}$, we define a *posting* as a tuple containing the following information:

- pre(u) is the position of u in $\mathcal{D}$;
- dist(u) is the sum of the insertion costs of all ancestors of u in $\mathcal{D}$, which is computed by means of the insertion distance;
- bound(u) is the position of the rightmost leaf of the subtree of $\mathcal{D}$ rooted at u;
- rencost(u) represents the renaming cost of the pattern tree node that matches u,
- embcost(u) stores the cost of embedding a pattern subtree into the subtree of $\mathcal{D}$ rooted at u. The value is zero if u is the match of a pattern tree leaf.
- embtree(u) stores the pattern subtree embedded into the subtree of $\mathcal{D}$ rooted at u whose cost is embcost(u).

The first three fields are computed when building the data tree of the considered XML document and they are not influenced by the execution of the matching algorithm. On the other hand, the last three fields are computed by the matching algorithm and may change during its execution.

We now define a data tree *index* [2] which contains an entry for each label occurring in $\mathcal{D}$. An entry for a label l contains

[2] For the sake of efficiency, the implementation indeed uses two indexes to encode the data tree: the former to store the leaf nodes of the data tree (i.e. the plain text elements) and the latter to store the inner nodes (i.e. the XML tags).

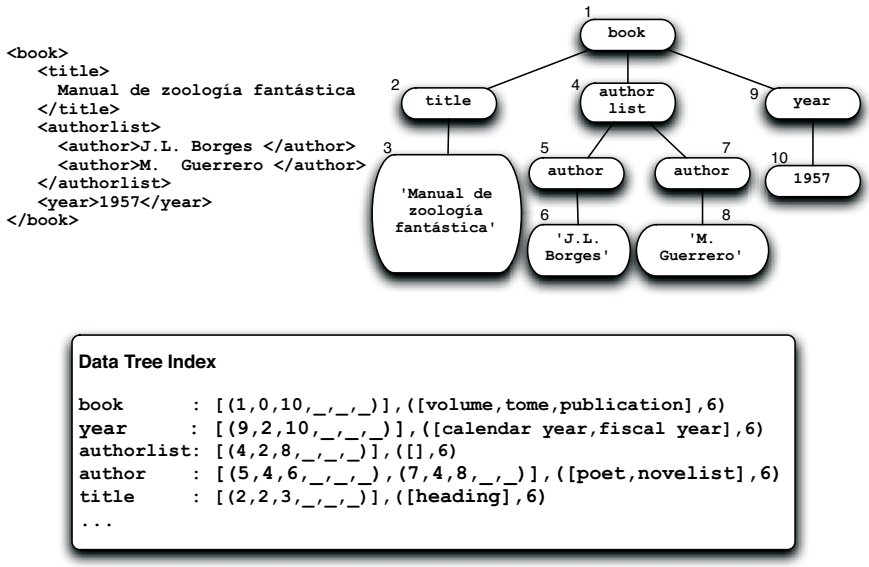


Figure 2. Data tree and data tree index for an XML document

- a list of all postings referring to nodes in $\mathcal{D}$ with label l. Such a list of postings is sorted by ascending preorder numbers.
- a pair containing a list of all synonyms of label l and the associated renaming cost;

Figure 2 illustrates the data tree $\mathcal{D}_{book}$ and the corresponding data tree index for a piece of XML. For the sake of readability, we omitted the postings regarding the leaf nodes of $\mathcal{D}_{book}$. Moreover, we assumed an insertion cost equal to 2 and a renaming cost equal to 6.

Typically, it is preferable to preserving information rather than removing it. Therefore, in order to tune up our filtering system, we considered the following rules of thumb:

- the deletion cost should be two or three times greater than the insertion cost to allow up to two or three insertion operations before preferring to delete a node;
- the renaming cost should be smaller than the deletion cost.

4.2 Expanded Pattern Tree

The expanded representation of a pattern tree allows one to explicitly encode all possible node deletions of a pattern tree node. More precisely, all permitted deletions of inner pattern tree nodes are represented by transforming the original pattern tree in the following way.

Every inner pattern tree node (except the root) w which represents a tag selector or a "," operator (i.e. boolean conjunction)[3] is replaced by a fresh binary "|"-

[3] Actually, in our system, we implemented an equivalent, optimized version of the pattern tree expansion

labelled node w_{or}. The left child of w_{or} refers to w, while the right child represents the fact that the pattern tree node is removed from the pattern tree. Basically, the fresh "|"-labelled (i.e. boolean disjunction) nodes inserted play the role of *choice points* in which the algorithm is called to decide whether to delete a node. Figure 1(b) shows the expanded version of the pattern tree depicted in Figure 1(a). Observe that we do not have to directly manage leaf deletions, which are hardcoded in the algorithm. Since leaves cannot be renamed, whenever a leaf node n of a pattern tree does not match any leaf node of the data tree, the pattern tree is automatically transformed by deleting n. Besides, the corresponding deletion cost is computed.

Now for every node w of the expanded pattern tree, we define a function delcost(w) which computes the total cost of deleting the node w and all its inner descendants. The returned value is greater than zero only for those right children of "|"-labelled node representing the deletion of an inner node.

In the following, sometimes an expanded pattern tree is simply called pattern tree.

4.3 Evaluating an unconditional, positive, ground filtering rule

For the sake of simplicity, we first describe how the methodology works on an unconditional, positive, ground filtering rule, and then we will describe how to implement the other language features. Basically, the problem amounts to finding one or all best approximate matches (i.e. also called embeddings) of a ground pattern against an XML document. We assume to have already generated both the expanded pattern tree $\mathcal{P}$ and the data tree index $\mathcal{D}$ representing the data tree.

The evaluation algorithm is based on the dynamic programming principle. The embedding cost of a subtree of the expanded pattern tree $\mathcal{P}$ rooted at a node w is calculated from (i) the embedding costs of the subtrees rooted at the children of w; (ii) the insertion distance between the match of w and the matches of the children of w.

All matches of pattern tree node labels against data tree node labels are stored in posting lists. To find the best approximate embedding of $\mathcal{P}$ in $\mathcal{D}$, the algorithm uses operations on postings. Two types of operations are needed.

(i) Given a posting list of potential ancestors and a posting list of potential descendants, the algorithm must find all ancestor-descendant pairs with the smallest embedding cost (*vertical axis*). Such an operation can be performed basically using some information stored in the posting list. Given a posting list S, by S_p, we denote the p-th posting of S. Moreover, $S_{[i,j]}$ represents the sublist of S containing the postings from position i to position j. If U is a posting representing a potential ancestor node u and R is a posting list including n potential descendants of u, recalling that the postings are ordered by preorder numbers, all the n descendants $v_1, \ldots, v_n$ of the node u must reside in the interval $R_{[j,j+n]}$ of the posting list R, since $\text{pre}(u) < \text{pre}(v_i) \wedge \text{bound}(u) \geq \text{pre}(v_i)$. When no

which only replaces tag selector nodes, while boolean conjunctive nodes are implicitly managed by the pattern matching mechanism.

deletions are allowed, the smallest embedding cost of u w.r.t. the descendants in R is thus calculated using the following formula

$$\begin{aligned}\text{embcost}(u) = &\min\{\text{embcost}(R_k) + \text{dist}(R_k) \mid j \leq k < j + n\} - \\ &\text{dist}(u) + \text{rencost}(u) + c_{ins}\end{aligned} \quad (1)$$

[4]

where c_{ins} is the insertion cost of a node. Whenever we also allow deletions, the smallest embedding cost become the minimum between delcost(u) and the value computed by the formula (1).

(ii) Given two posting lists that represent the embeddings of two distinct children of a pattern tree node w, the algorithm must find all pairs that belong to the same data node (*horizontal axis*).

In this case, if w is connected to its children through a boolean conjunction, then the sum of the embedding costs must be calculated, whereas w is connected to its children via a boolean disjunction (i.e. "|" or "?" operator), then we must select the embedding with the cheapest cost.

When evaluating a ground filtering rule, the algorithm visits the pattern tree nodes in depth-first order. During the traversal, it fetches from the data tree index the posting lists belonging to the labels of the visited nodes. Arrived at the leftmost leaf, it joins the posting belonging to this leaf with the posting belonging to the parent node (*vertical axis*) and then proceeds the visit. If a node u has two or more children, then the cheapest combination of matches belonging to u's children is chosen and stored in the posting list generated for u (*horizontal axis*).

The posting lists returned by the algorithm contains the postings representing the transformed pattern that best matches the data tree along with the embedding cost.

4.4 Evaluating a generic filtering rule

As shown in Section 2, a filtering rule is a quite complex object which may contain non ground patterns and filtering conditions. Moreover, according to the filtering operator and the filtering mode chosen, it can be employed for approximate/exact matching and to implement positive as well as negative filtering. In the following, we briefly discuss how to adapt the algorithm presented in the previous section to cope with such features.

Nonground patterns and filtering conditions.

Patterns may contain variables. Therefore, substitutions that bind such variables to subparts of the data tree must be computed during the matching process to find the desired embeddings.

We extend the matching algorithm to deal with variables in the following way. Given a nonground pattern tree $\mathcal{P}$, we consider the pattern $\mathcal{P}'$ which is obtained

[4] Given a data tree node v, which is represented by the posting R_k, embcost(R_k) (resp., dist(R_k)) stands for embcost(v) (resp., dist(v)).

from $\mathcal{P}$ by removing all the variables in $\mathcal{P}$. For each node labelled with a variable we removed, we record its position into a list. Since $\mathcal{P}'$ is ground, we can apply the previous tree matching algorithm to find the embeddings of $\mathcal{P}'$ into the data tree. Moreover, as we saved the positions of the variables appearing in $\mathcal{P}$, we can compute the embedding substitutions by analyzing the matched subtrees in the data tree and hence selecting those parts which correspond to the variable positions. It can happen that a variable cannot be bound to a subtree, e.g. the algorithm computes an embedding for $\mathcal{P}'$ which requires some node deletions involving an ancestor of a variable node. In this case, such an embedding is simply discarded.

By using this approach, we can thus produce all the possible embedding substitutions for a nonground pattern $\mathcal{P}$ by simply analyzing all the embeddings of $\mathcal{P}'$ in the data tree. We can then apply such substitutions to filtering conditions to generate instantiated conditions and subsequently check their satisfiability.

Approximate and Exact Filtering.

The algorithm we described is mainly used for approximate filtering. Nevertheless it can be employed for exact matching. We just have to look for matches with a null embedding cost, since no insertion, deletion, or renaming of nodes is allowed in this case. Therefore, to optimize the search one can think to ignore the basic pattern transformations. Unfortunately, the insertions are implicitly defined in the matching algorithm, and hence we cannot avoid them. However, deletions and renamings can be disabled in the following way: (i) to avoid deletion of nodes, we simply use the original pattern tree instead of using its expanded representation; (ii) to avoid renaming of nodes, the synonymity relation encoded in the data tree index is ignored.

Positive and Negative Filtering.

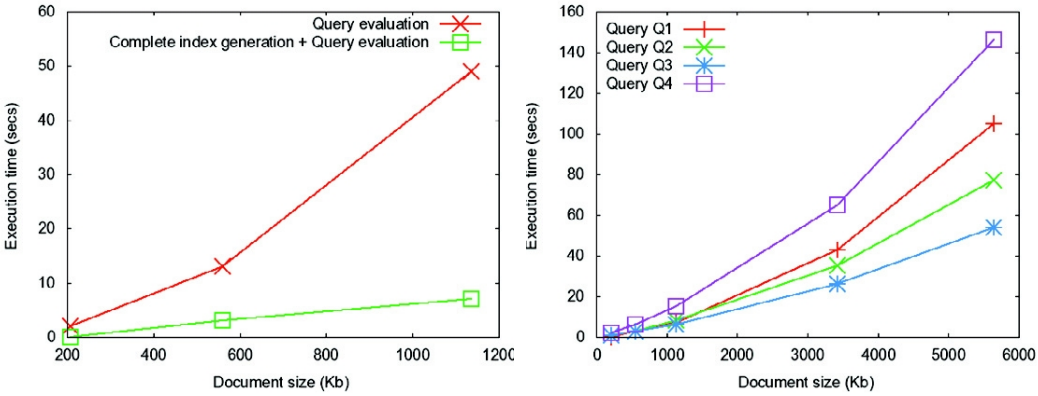
As shown in Section 2, our language defines syntax constructs for negative filtering which incorporate the expressive power of negation into our formalism. To enforce such a behaviour, we just execute the negative filtering rule as it was a positive one. We then remove all the embeddings found from the data tree returning the desired filtered outcome. In order to exactly remove all and only the desired information, we allow only to "filter out" exact matches. In other words, any negative filtering is always an exact filtering.

Since negative filtering needs a positive filtering execution along with an entire data tree traversal for removing the embeddings found, it follows that the implementation of negative filtering is less efficient than the implementation of positive filtering (see Section 5 for further details).

5 A Lazy Implementation: an Experimental Evaluation

The proposed filtering language has been implemented in the Phil system, which is freely available at http://www.dimi.uniud.it/demis/#software. The implementation has been written in the lazy functional language Haskell with the aim

(a) (b)



Document size (Kb)	Ex. time: case 1 (s)	Ex. time: case 2 (s)
208	2.459	0.930
560	13.227	3.001
1136	49.212	7,492

Document size (Kb)	Ex. time Q1 (s)	Ex. time Q2 (s)	Ex. time Q3 (s)	Ex. time Q4 (s)
208	0.985	1.267	1.101	2.183
560	3.046	3.941	3.158	6.812
1136	7.781	8.428	6.647	15.196
3424	43.053	35.248	26.674	65.821
5640	105.358	77.766	54.801	146.839

Figure 3. Experimental evaluation of the PHIL System

of showing how *laziness* can be particularly fruitful in developing such kind of applications. Lazy functional (and functional logic) languages allow one to somehow "minimize" the amount of information needed to be processed in order to evaluate expressions.

As we have seen in Section 4, our methodology employes a quite sophisticated data structure whose whole generation may be very time-expensive. Therefore, exploiting laziness in this context amounts to saying that only the portion of the data tree index which is strictly necessary to evaluate a filtering rule is generated with a consequent gain in the overall system performance. Our claim is supported by the following experiment. We have evaluated a given filtering rule on XML documents of increasing sizes in two distinct ways.

(i) We have forced the whole data tree index generation before executing the rule.

(ii) We have just "lazily" executed the given filtering rule letting Haskell to produce the (portion of) the data structure which is needed to process the rule.

The results of our experiment are depicted in Figure 3(a) and clearly show how laziness speeds up the query evaluation by automatically avoiding the construction of unnecessary data structure (e.g. for a of 1Mb XML document, the evaluation of the filtering rule in Case 2 is ∼7 times faster than the rule evaluation of Case 1).

Qualitative and Quantitative Analysis.

In order to evaluate the usefulness of our approach in a realistic scenario, we have benchmarked our system using the XMark benchmark suite[5]. The suite offers a set of 20 queries, each of which is intended to challenge a particular primitive of the filtering engine, along with the XML documents generator xmlgen which can be used to produce the synthetic data on which running the experiments. By means of our formalism, we are able to express 17 queries out of 20. The remaining 3 queries cannot be formalized, since they involve document transformation and computational capabilities which are out of the scope of a simple filtering language (e.g. lexicographic ordering (query n. 19), currency conversion (query n. 18), and output formatting (query n. 10)).

From a purely quantitative point of view, we tested the system on a Macbook Intel Core 2 Duo 2Ghz equipped with 2Gb of RAM memory. We defined 4 filtering rules encompassing all the language features. Specifically, rule $Q1$ models a positive nested filtering rule with regular expressions, rule $Q2$ is a positive nested filtering rule which includes applications of the occurrence operator and arithmetic tests, rule $Q3$ employs the counting operator, and finally rule $Q4$ is an example of negative filtering. All the considered rules contain one or more occurrences of the synonymity operator.

Figure 3(b) shows the results obtained by executing the 4 filtering rules to 5 different, randomly generated XML documents which have been synthesized by xmlgen data generator. We tuned the generator in order to yield XML documents whose size ranges from 208Kb to almost 6Mb. Execution times of each filtering rule are computed as the average of three filtering rule's runs. The preliminary results are quite encouraging even on experiments that exceed the toy size: all the rules are evaluated in less than 3 minutes on a repository whose size is almost 6Mb. Moreover, it is a matter of few seconds obtaining an answer on a 1Mb XML document. Finally, note that negative filtering behaves worse than positive filtering. This is mainly due to the fact that, in the current implementation, negative filtering has to traverse the entire data tree (which may easily consist of more than 100000 nodes) in order to get rid of the detected patterns. Therefore, when dealing with large XML documents, the overhead due to the data tree traversal might be considerably high.

6 Conclusion

The growing complexity of the World Wide Web demands for tools which are able to tame the so-called information overload. To this respect, filtering and query languages allow one to extract relevant and meaningful information within the enormous amount of data available on the Web. In this paper, we firstly presented a declarative XML filtering language which has several advantages w.r.t. other approaches. It is inspired by the approximate pattern-based query language ApproXQL and extends it by introducing a number of new syntax constructs which provide a much more expressive framework (e.g. negative filtering, pattern variables, nested queries, conditional filtering, etc.). Secondly, we implemented the language

in the prototype PHIL using the lazy functional language Haskell pointing out the inborn benefits of laziness by means of a thorough experimental evaluation.

Finally, let us conclude by mentioning some directions for future work. We are currently formalizing a denotational semantics of the language which precisely models the behaviour of the language constructs. Moreover, we are trying to combine the filtering language with an ontology reasoner which simplifies and automatizes the retrieval of the synonymity relation by querying a given (possibly remote) ontology.

References

[1] M. Alpuente, D. Ballis, and M. Falaschi. Automated Verification of Web Sites Using Partial Rewriting. *Software Tools for Technology Transfer*, 8:565–585, 2006.

[2] D. Ballis and D. Romero. Filtering of XML Documents. In *Proc. of 2nd Int'l Workshop on Automated Specification and Verification of Web Systems (WWV'06), Paphos (Cyprus)*, pages 19–26. IEEE Computer Society Press, 2006.

[3] M. Bezem. *TeReSe, Term Rewriting Systems*, chapter Mathematical background (Appendix A). Cambridge University Press, 2003.

[4] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. Technical report, 2002. Available at: http://www.xcerpt.org.

[5] Centrum voor Wiskunde en Informatica. XMark – an XML Benchmark Project, 2001. Available at: http://monetdb.cwi.nl/xml/.

[6] A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and Abstract Semantics of a Graphical Query Language. *Theoretical Computer Science*, 275:521–560, 2002.

[7] P. Kilpeläinen. Tree Matching Problems with Applications to Structured Text Databases. Ph.d. thesis, University of Helsinki (Finland), 1992.

[8] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 2004.

[9] The Open Group. Unix Regular Expressions. Available at: http://www.opengroup.org/onlinepubs/7908799/xbd/re.html.

[10] S.Amer-Yahia, M. F. Fernández, D. Srivastava, and Y. Xu. Phrase matching in xml. In *Proc. of 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 177–188, 2003.

[11] T. Schlieder. ApproXQL:Design and Implementation of an Approximate Pattern Matching Language for XML. Technical Report B 01-02, Freie Universität Berlin, 2001.

[12] T. Schlieder and H. Meuss. Querying and Ranking XML documents. *Journal of the American Society for Information Science and Technology JASIST*, 53(6):489–503, 2002.

[13] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: http://www.w3.org.

[14] World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: http://www.w3.org.

[15] World Wide Web Consortium (W3C). XQuery: A Query Language for XML, 2001. Available at: http://www.w3.org.