Order-Preserving Optimization of Twig Queries with Structural Preferences

SungRan Cho Wolf-Tilo Balke
L3S Research Center
University of Hannover, Gemany
(+49) 511 762-17712
{scho, balke}@l3s.de

ABSTRACT

Efficient query processing using XPath or XQuery has inspired a lot of research. In contrast to classical exact match retrieval, in today's systems, specifying preferences rather than simple hard constraints is essential. As the structure of XML documents plays a major part in retrieval, recently approximate query matching on structure has received attention. However, query processing of structural user preferences has not yet been considered. In this paper we enable users to express structural preferences and consider the problem of optimizing XML twig queries while preserving the ordering induced on the result set by such user preferences. Evaluating such queries generally needs a rewriting into a set of queries, where each leaf node can be expanded by combinations of structural elements derived from the preference information. Since such structure expansions typically contain redundancies and the efficiency of query evaluation strongly depends on the size of the set of rewritten queries, it is important to identify and simplify necessary expansions. We give a detailed analysis of this process and present an optimization algorithm that determines a minimal set of queries, which in turn are minimal in their expanded nodes, while maintaining the ordering induced by the preference structure. Finally, we provide a comprehensive practical evaluation of our optimization against the XMark benchmark dataset.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval – *query formulation, retrieval models, search process.*

General Terms

Algorithms.

Keywords

Preference-based retrieval, XML databases, personalization.

1. INTRODUCTION

Today XML is ubiquitous in retrieving and exchanging information over the Internet. All kinds of structured information, like catalogs for e-commerce applications, can be expressed in XML documents. Such documents are queried using advanced retrieval languages like XPath or XQuery taking into account both: the values of data items and the structure of the document they are found in. Like the XML documents, queries are usually structured to express the user's information needs. Users often have specific preferences about the structure. Especially if the document structure shows a certain semantics (often described by a DTD), evaluating queries with such structural preferences is necessary. In this paper we focus on this important kind of queries, so called *twig queries with structural preferences*.

When querying for information users may have a vague idea of what kind of information they look for as well as where it occurs in a specific XML document.In this context, personal preferences express user wishes that might not always be fulfilled, but allow for relaxation of query predicates. Preferences are becoming an important paradigm in query processing allowing for personalization with respect to a user's interests. Let us consider a short example on querying e-catalogs that we also use for illustrational purposes in the remainder of the paper: given an e-catalog about media we might have a DTD as shown in Figure 1(a). In our example a catalog can have items as books, software, or CDs and information about the delivery. Users usually have strong preferences on whether to buy books or for instance rather audio books on CD, or software like games. A typical query could be: ``retrieve all media matching the keyword 'Star Wars'. I would prefer books, over CDs or games". Since users do not really know, what documents can be expected in the collection, most query predicates like the media type do not have a single specified value, but rather a set of more or less preferable alternatives for each value, thus inducing a specific ordering on the result set.

Such human preferences in query processing have recently gotten considerable attention. [1-3] define frameworks on how to model preference queries in database retrieval using partially ordered preference graphs. The notion of a preference is usually considered as a subsumption hierarchy [3], where query processing is stopped after returning all matches that are optimal, e.g., with respect to the Pareto semantics. In document retrieval and Web search engines it has been always considered rather helpful to retrieve an ordered list of documents that best reflect the user preferences for preference-based scorings (see, e.g. [4] for an example on e-catalogs).

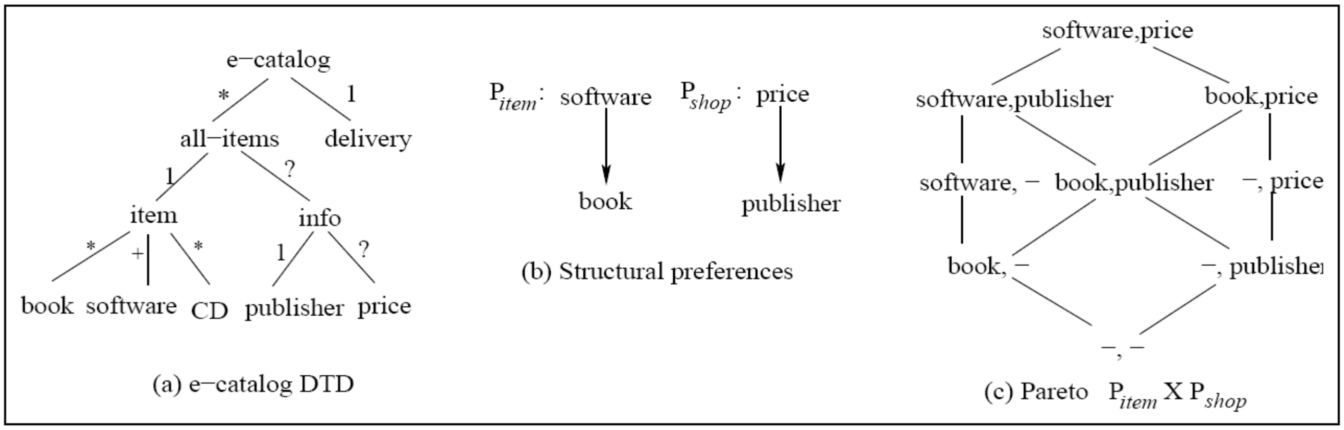


Figure 1: Example DTD and preferences

But up to now preference queries did only consider values, whereas in XML queries also preferences on element tags are valid, i.e., preferred tags become part of the query structure. We also understand structural preferences here in the way of result order constraints. That means a preference twig query is rewritten in a set of queries progressively posed to the database: starting with a query expanded with all top attributes stated in a user's preferences, each predicate is gradually relaxed to some less preferred attributes until finally the general query without any preference attribute is posed (e.g., using the scheme given in [5]). Therefore, if the query contains multiple preferences $P_1,\ldots,P_n$ and each preference P_i states a number of $|P_i|$ distinct attributes, the total number of queries posed is $|P_1+1| *\ldots* |P_n+1|$.

In XML query processing the extension of traditional exact match models has been heavily discussed within the community. But, in contrast to the very advanced research on optimizing conventional XPath or XQuery queries, optimizations for XML preference queries are still lacking. In this paper, we first present a framework for expanding twig queries with structural preferences expressed by the user. Since such structure expansions typically contain redundancies and the efficiency of query evaluation depends on the size of the set of rewritten queries, it is important to identify and simplify necessary expansion queries for effective evaluation. Thus, our challenges are to minimize expansions of a twig query with structural preference information while preserving the order of the result set. Furthermore, we minimize preferred elements in every single expansion query. As we will show in this paper there is an essential gain in optimizing the queries with preference constraints before they are posed. Our technical contributions are the design of a complete optimization framework for twig queries with structural preferences and the development of an efficient algorithm to carry out the optimization.

2. PROBLEMS OF STRUCTURAL XML RETRIEVAL

In this section we investigate the problem of preference-based structural XML retrieval and motivate our solution. First consider a few examples to obtain an intuition about what an order preserving preference optimization has to deal with. We will specifically focus on minimizing the number of necessary expansions of a given twig query.

Example Scenario: The DTD given in Figure 1(a) represents the XML database schema of an *e-catalog* containing information about each item on sale and delivery information. Edges between elements are labeled with '+' (one or more), '*' (zero or more), '?' (zero or one), or '1' (exactly one) to indicate the frequency of subelements. In the example DTD, the elements *all-items*, *book*, *CD*, *info*, and *price* under their parents are optional. All other elements are required under their parents.

In database retrieval users will generally express certain preferences. Such preferences state what a user likes/dislikes and preference query processing attempts to retrieve all best possible matches in a cooperative fashion avoiding empty result sets. Usually such preferences are specified over data values (e.g., a user might prefer the lowest price), but in XML retrieval also the structure of the document often carries semantics. Our catalog is a typical example: all objects on sale are categorized by using document structure. The sample preference graphs of Figure 1(b) show two different user preferences about the structure, in which P_{item} expresses that *software* is preferred over *book*, whereas P_{shop} states that *price* is preferred over *publisher*.

To specify such preference constraints in XML queries we will mark query nodes with preferences associated with a preference graph. Example queries marked with structural preferences are depicted in Figure 2. We will focus on twig queries, in which nodes are labeled with either element tags or values, and edges are either parent-child (single edge) or ancestor-descendant (double edge), while a dashed edge is between an element and a string value. In each query, some nodes are marked with '*', indicating that these nodes are returned as answer (i.e., projected), and only leaf nodes can be annotated with structural preferences.

Unfolding of Preference Queries: We use the term expanded *node* to denote a new node added to a query. A preference twig query is *unfolded* by adding nodes taken from some preference graph to the query before evaluation. Edges to expanded nodes are always generalized to ancestor-descendant relationships allowing matchings to all relevant portions of the XML database.

Preferences induce a ranking on the result set. Let us now consider the order induced by multiple preference attributes. An order of attributes is enumerated by successively relaxing query predicates from most to least preferred attributes in each of the preferences,

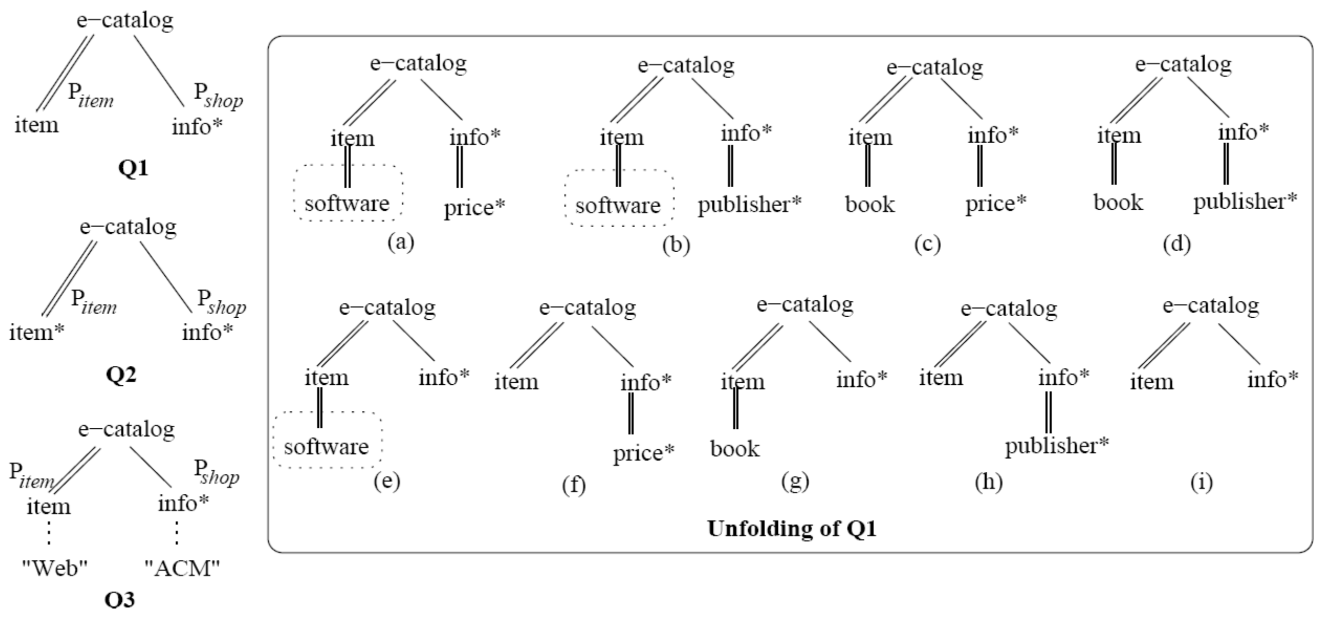


Figure 2: Example twig queries with preference constraints

considering all variations as query rewritings, and collecting the results with respective rank information (e.g., as shown in [5]). If we do not have an ordering (or weighting) of the preferences the *Pareto semantics* will provide a fair relaxation scheme. For example, Figure 1(c) shows the Pareto order induced by two preferences P_{item} and P_{shop} of Figure 1(b): the *software*, *price* combination is the most preferred match and next are the equally preferred *software publisher*, match and the *book*, *price* match (each relaxing a single preference). The least preferred match is any combination of items (totally dropping both preferences). Given query Q1 in Figure 2 containing two preferences P_{item} and P_{shop}, this induced Pareto order is reflected by all possible expansions of Q1 in Figure 2(a)-(i) in the unfolding process. Starting with query (a), whose answer set the user prefers most, over queries (b) and (c) that are equally preferred in second place, down to query (i) as the least preferred option.

Whenever preference marked nodes are distinguished nodes, their expanded nodes inherit the status of projection. The intuition behind this is that answers should contain the projected nodes, but should also show in how far the users' preferred nodes could be granted. For this reason, the twig queries in Figure 2(a)-(i) additionally project *price* and *publisher* nodes under each *info* node.

Optimization of Structural Preference Queries: Preferences in XML queries generally can be composed of either values or element tags. If it is about *values*, unfoldings cannot be reduced, because in principle all values are possible matches for a query (cf. [6]). For preferences on *element tags*, however, the query structure and the respective DTD can help to prune the rewritten queries encountered in the unfolding. However, because it is necessary to preserve the order induced by the preferences, we cannot simply look for query containments like for example in [7]. But if the preferences at different nodes in the original query are redundant, pruning irrelevant nodes or entirely removing redundant rewritings can lead to huge improvements in query execution times.

(a) Removing Irrelevant Expanded Nodes: Given a preference on preferred element tags, we know from the DTD what structural elements can be possibly encountered beyond a certain node. Thus, a simple check against the DTD can help to reduce nodes in some preferences and we end up with a smaller number of possible expansions (and thus rewritten queries), before we even start expanding query nodes.

(b) Removing Redundant Expanded Nodes: But even if all expanded nodes are relevant, not all rewritten queries have to be posed against the data collection. Some of them retrieve redundant results. We next study which nodes or queries can be eliminated by again taking advantage of DTD information; to be more specific by testing edge labels in a DTD path. Consider the twig query Q1 and take a closer look on the expansions of Q1 in Figure 2. In the twig query (a), the expanded *software* node under the *item* node can be reduced because the node is not a *distinguished* node and we find that it is always *required* by examining the corresponding *software* node from the DTD of Figure 1(a) (the path in the DTD from *item* to *software* has edge label '+'). The same happens for queries (b) and (e). However, the *book* node of queries (c), (d), and (g) cannot be eliminated because it is optional.

In contrast, consider the similar looking query Q2 in Figure 2, where the query projects both *item* and *info* nodes. The expansions of Q2 are the same as Q1's, i.e., Figure 2(a)-(i), with the exception that *item* and its expanded nodes are distinguished nodes. In this case, we cannot prune the *software* node of queries (a), (b), and (e) since distinguished nodes can never be redundant. Moreover, the twig query Q3 in Figure 2 again looks similar, but here the *item* and *info* nodes are constrained by different values. Since nodes with preferences are expanded between the node and its associated value, all the expansions of Q3 have to be kept due to the constraining values.

(c) Removing Nested Queries w.r.t a Preference Order: We refer to a twig query Q' as being *nested* in a twig query Q, if the query result of Q' is a subset of the query result of Q. Since nested queries are redundant, we can usually remove all nested queries from the set of all expansion queries. However, while our focus is to obtain a minimal set of query expansions, there is a crucial difference: nested queries can only be pruned, if that does not violate the induced preference order. After the previous optimization all individual expansion queries havea minimal set of expanded nodes. But take a closer look at the queries expanded from the twig query Q1 in Figure 2. After removing the redundant *software* node, query (a) is equivalent to query (f). Similarly, the optimized versions of (b) and (e) are the same as (h) and (i), respectively. Since the queries (f), (h), and (i) are redundant and *less preferred in the induced order*, we can eliminate them.

However, the set of Q1's expansions is still not minimal. The result retrieved by query (g) is contained by query (e). Thus, we can drop (g). Additionally we find that the query (e) is a subquery of (b) by testing that all differing nodes with respect to the superquery (b) are required in the DTD in Figure 1(a) Furthermore, the surviving queries (a), (b), (c), and (d) still are not minimal: we can see that the results retrieved by queries (c) and (d) are contained in those by queries (a) and (b), respectively. After dropping (c) and (d), the queries (a), (b) without the *software* node cannot be reduced any further and are thus minimal while preserving the induced preference order. A similar optimization applies to the query Q2. Here we can obtain the minimal query expansions (a), (b), (c), and (d), because the rest are nested in these queries.

We have now gained a good intuition about what optimizations apply in each individual query unfolding. Let us now state the optimization problem:

Optimization Problem: Given a preference twig query Q and a tree-structured DTD D, find a minimal set of order preserving query expansions of Q with a minimal set of expanded nodes on D.

3. FORMALIZATION OF OPTIMIZATION SEMANTICS

In this section, we formalize the semantics of order-preserving twig query optimization with preferences. In particular, we will consider tree-structured DTDs, partial order preference graphs, and twig queries.

Definition 1 [DTD Tree]: A *DTD tree* is a rooted node-labeled and edge-labeled graph T = (V, E), such that:
(i) each node in V is labeled by an element tag, and
(ii) each edge has a label from the set {'+', '*', '?', '1'}.

In the DTD, the multiplicities specify whether and how subelements of an element can repeat. These are labeled in edges of the DTD: '?' for zero or one, '1' for one, '*' for zero or more, '+' for one or more.

Definition 2 [Preference Graph]: A *preference graph* is a finite node-labeled directed graph P = (V, E), such that:
(i) each node in V is labeled by a value,
(ii) the direction of each edge between nodes in P expresses that the attribute of the label, where the edge originates, is preferred over the attribute of the label, to which the edge point

Definition 3 [Preference Twig Query]: A *preference twig query* is a rooted node-labeled and edge-labeled tree such that
(i) its nodes are labeled with unique element tags, with the exception that its leaves may be labeled by element tags or values,
(ii) its edges are labeled parent-child or ancestor-descendant,
(iii) some element nodes are marked as distinguished and subsequently returned as query result, and
(iv) some element leaves are marked with preferences

We assume that original queries are minimal (i.e., the smallest size) and all query nodes are unique. Although it is desirable to account for minimizing the query while new nodes are added, a detailed discussion of this direction is beyond the scope of the paper. Our focus is only expanded nodes to be reduced, which facilitates obtaining a minimal set of expansion queries. Preferences can only occur in leaf nodes, since that is an appropriate part where user preferred elements take place more specifically. Figure 2 shows some examples of different preference queries. A parent-child relationship is shown as a single line, whereas ancestor-descendant relationships are given by double lines. For evaluation consider the following XQuery path expression of query Q1 in Figure 2 with two preferences P_{item} and P_{shop} incorporated:

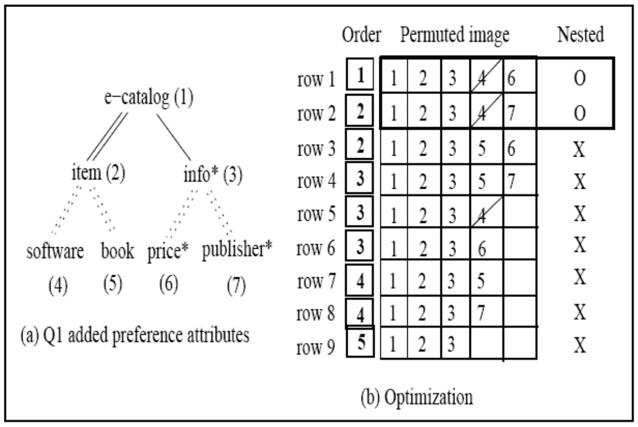
sync(e-catalog[.//item[**preference**(P_{item})]]/info[**preference**(P_{shop})])

We implement a preference at a node by a *preference* function, with e.g., P_{item} as an input argument. In the unfolding process the function will expand the marked query node by attributes in P_{item}. The *sync* function interacts with multiple preference functions given in the query and computes their conjunction following the Pareto semantics. A preference query needs to be rewritten as an ordered set of queries, retrieving all data relevant to answering the original query. We thus have to look at all possible relaxations of the user preferences and expand the preference marked nodes accordingly. Let us first define the case of a single preference in a query:

Definition 4 [Unfolding of a Preference Query]: Given a twig query Q containing a node n marked with preference P, an expansion query Q' is obtained by *unfolding* Q:
(i) adding a node v of P to a query node n as a successor (in the case where n is value-constrained, v resides between n and the value), adding an edge with ancestor-descendent label from n to v, and propagating a possible distinguished status of n down to v,
(ii) entirely removing P from Q.

This results in |P+1| distinct queries, in which the most preferred query is expanded by (one of) the most preferred node(s) in the preference graph and the least preferred query of the expansions is obtained by simply removing mark P from query node n (i.e. the maximum possible relaxation of the preference, since preferences cannot always be fulfilled). All queries of expansions are conventional twig queries, which can be posed against any XPath or XQuery evaluation engine. The generalization of Definition 4 to the case of multiple nodes marked with preferences is straightforward: each single preference is expanded individually such that every possible combination of preference attributes is reflected. Given a preference twig query and a DTD, we now have to minimize expansions of the query in a way that preserves an ordering induced by the preference. Some expansion queries can be eliminated because they are equivalent or contained in other expansions:

Definition 5 [Nested Query]: A query Q1 is *nested in* a query Q2 (Q1 $\subseteq$ Q2), iff for every XML database D, Q1(D) $\subseteq$ Q2(D).



	Order	Permuted image					Nested
row 1	1	1	2	3	~~4~~	6	O
row 2	2	1	2	3	~~4~~	7	O
row 3	2	1	2	3	5	6	X
row 4	3	1	2	3	5	7	X
row 5	3	1	2	3	~~4~~		X
row 6	3	1	2	3	6		X
row 7	4	1	2	3	5		X
row 8	4	1	2	3	7		X
row 9	5	1	2	3			X

Figure 3: Optimization of Q_1

Definition 5 also implies that the order of subtrees in a query is immaterial, simply because the same results are returned. Given a preference twig query Q, a set of expansions of Q is *minimal*, if there is no nested query in the set.

4. OPTIMIZING PREFERENCE TWIG QUERIES

After having formalized the optimization steps, in this section we develop an algorithm for obtaining a minimal set of order-preserving expansions. Analyzing the structure of all possible queries in the unfolding of the original preference query, our algorithm uses three distinct analysis techniques.

- *A preference attribute analysis:* it removes irrelevant preference nodes using the DTD information, which reduces the size of a set of expansions from the unfolding.
- *An expanded node analysis:* it removes redundant expanded nodes by analyzing DTD paths, which reduces the size of each individual query from the unfolding.
- *A pair-wise analysis of expansion queries:* it removes nested queries, based on containment/equivalence relationships between queries with respect to the preference order, which reduces the number of expansion queries from the unfolding.

Before we discuss all optimizations in detail, the pseudocode in Figure 4 shows the complete algorithm and the necessary procedures.

4.1 Preference Analysis

This analysis compares preference graphs and DTDs to determine, if all preferred attributes really can be part of a correct query. Given a twig query Q, for each query node n marked with preference P, we check that nodes in P also occur in the DTD: traverse the graph P and for every node in P, check if that node is a valid descendant of n in the DTD. If the node is valid, add it to n as a descendant (if n is constrained by a value, the node is added between n and the value). If a preference graph does only contain invalid elements, entirely remove preference P from n. This is performed by the procedure *add-node* in our algorithm *OptiPref*.

```
Algorithm OptiPref
Input: query Q, DTD tree D;
Output: minimum expansions of Q along with
            minimum expanded nodes in each expansion;
let PQ be the permuted image of Q from the most
to the least preferred order;
/* Q is traversed in a right to left direction */

for (n is leaf node of Q with a preference) {
   add-node(n, D); }
for (m is expanded node of Q) {
   if (m is not projected and required(m)=TRUE) {
    remove(m,PQ); //remove m from PQ
}}
remove-nestedQuery(PQ);
/* build optimized expansions of Q for evaluation */

unfoldQuery(PQ);
end Algorithm

procedure add-node
Input: leaf node n with its preference graph pg;
Method: add nodes in pg to n using the DTD;
let n',v' be the DTD graph nodes corresponding to nodes n,v, resp.;
for each node v in pg of n {
   if (∃ a DTD path from n' to v') {
      add v to n; //with ancestor-descendant edge
      if (n.value ≠ null) {
         v.value = n.value;
         n.value = null;
         constrained(v) = TRUE;}
      else constrained(v) = FALSE;
      if (n is projected) projected(v)=TRUE;
      required(v) = required-node(n',v');
}}
end procedure

procedure remove-nestedQuery
Input: permuted images PQ;
for each i of $PQ_n, \cdots, PQ_1$ { //reverse order
   get last position k in $PQ$, s.t k.order=i.order;
   for each j(≠i) of $PQ_k, \cdots, PQ_1$ {
      if ($PQ_i$=$PQ_j$) {
         if ($i$ ¡ $j$) remove $PQ_j$;
         else remove $PQ_i$; }
      else if ($PQ_j \subset PQ_i$ and $\forall v \in (PQ_i - PQ_j)$,
          projected(v)=FALSE) remove $PQ_i$;
      else if ($PQ_i \subset PQ_j$ and $\forall v \in (PQ_j - PQ_i)$,
          required(v)=TRUE and constrained(v)=FALSE)
         remove $PQ_i$;
}}
end procedure

function required-node
Input: DTD nodes v', w';
Output: TRUE if w' is required;otherwise FALSE;
if (∃ a DTD path p from v' to w' and
       ∀edge e on p, s.t label(e)=1 or +) return TRUE;
else return FALSE;
end function
```

Figure 4: Algorithm `OptiPref`

To show the effects of this analysis let us for instance consider the preference twig query Q1 in Figure 2 with the preferences P_{item}

and P_{shop} of Figure 1(b) and the DTD of Figure 1(a). Since both *software* and *book* are possible descendants in the query from the DTD, we add them to the *item* node in Q1. Similarly *price* and *software* are added to the *info* node. Figure 3(a) shows the resulting query.

4.2 Expanded Node Analysis

Building a Permuted Image for Optimization: Optimizing query expansions includes the optimization of each individual query of the unfolding (e.g., redundant nodes) and the optimization of queries in the unfolded set (e.g., nested queries). Since the number of queries in a unfolding explodes for more complex preferences and many nodes marked with preferences, optimization techniques have to work on a suitable abstract representation of the query, called an *image*.

For our optimizations we use an enumerated image of the twig query. We first expand a single twig query with all possible nodes given by preferences and then assign unique numbers to every node (original and expanded) independently of the node's type. An unfolding of the query then can be seen as a set of sequences of numbers, where each sequence contains all the original nodes' numbers and a subset of the expanded nodes' numbers. Since we can independently relax every preference attribute in the twig query, all possible combinations of expanded nodes' numbers would have to be created for completely unfolding the query. The unfolded set can thus be seen as a set of sequences, which we refer to as a *permuted image*. Our permuted image adheres to the Pareto semantics and the ordering induced reflects multi-attribute ranking techniques.

For example, for query Q1 in Figure 2 and the four relevant nodes of two preferences P_{item} and P_{shop}, Figure 3(a) shows the respective enumerated image, i.e., original nodes (1, 2, and 3) and expanded nodes (4, 5, 6, and 7). The unfolding of Q1 is performed by permuting the expanded nodes as shown in Figure 3(b). Having the permuted image as representation of a query we can proceed with expanded node analysis.

Analyzing Expanded Nodes: for any query we have to find an equivalent query with the smallest number of expanded nodes. This analysis identifies, which expanded nodes are redundant and can be removed from a query. A query node n can be removed from a permuted image, whenever the following condition holds:

- n is not a distinguished node, and n is not constrained by a value, and n is *required*.

We will refer to the above condition as *eXn* (elimination of expanded nodes). The basic idea behind *eXn* is the following: if a query node n is a required node, we always encounter all data instances in any XML database. Thus, we can prune node n.

The correctness of *eXn* is obvious: due to both being expansions of Q, queries Q1 and Q2 can only differ in the expanded nodes, i.e., leaf nodes. Let us assume that $Q2(D) \subseteq Q1(D)$ holds for any database D, but any of the conditions in *eXn* is violated. If any node $n \in (Q1 - Q2)$ is distinguished or node n is constrained by a value 'x', constructing an example where Q1's and Q2's result sets differ is trivial. To show the necessity of the third part of *eXn* assume that node n would not be required. Now we will construct a document D2 containing two instantiations p1 and p2 of the parent node p of n, where only p1 contains a child node of type n. Again, whereas Q2 can perform a join between p1 and p2, Q2 would need a witness of type n to perform such a join and thus again contrary to the prerequisite the result set of Q2(D) is larger. Thus we have stated the set of conditions *eXn* as necessary to perform the removal of expanded nodes.

As an example of applying *eXn*, consider query Q1 and its permuted image in Figure 3(a) and (b). The *software* node (4) in rows 1, 2, and 5 in the permuted image (i.e., the queries in Figure 2(a), (b), and (e)) can be removed, because for node (4) condition *eXn* holds. Figure 3(b) shows the operation of eliminating redundant expanded nodes on the permuted image. As another example we have also already motivated the optimization of query Q2 in Figure 2. In Q2, no node is pruned, because all expanded nodes are distinguished nodes contradicting *eXn*.

4.3 Identifying Order-Preserving Unique Expansions

After the first two optimizations we have minimized expanded nodes for each individual query, but have not yet reduced the total number of queries in the unfolding. However, some of them may retrieve subsets of other queries' results, or even the same sets. The next step deals with removing equivalent and nested queries, i.e., if a query is a direct subset of another query, we can drop the subset query. This optimization again makes use of the permuted image by pairwise comparisons of sequences in the image.

For twig queries, a query Q2 is nested in a query Q1 (i.e., $Q2 \subseteq Q1$), if (i) Q2 preserves all node types of Q1, (ii) Q2 preserves all structural relationships of Q1 (a child/descendant edge of Q2 is mapped to a child/descendant edge) of Q1), (iii) Q2 preserves all '*' labels of Q1. Generally, whenever two queries are equivalent we may simply drop either query. Whenever a query is a proper subset of another query, we still need to identify whether its result is really nested in the other query's result, but if so can also simply drop the nested query. However, if we have to preserve a preference-induced order between the results of two queries Q and Q', we can only prune an equivalent or a nested query whenever one of the following conditions holds:

- if $Q \equiv Q'$ and order(Q) = order(Q'), then prune either Q or Q'
- if $Q \equiv Q'$ and order(Q) < order(Q'), then prune Q
- if the nodes and edges of Q' are a subset of nodes and edges of Q, and order(Q) $\leq$ order(Q'), and all differing nodes v in (Q – Q') are not distinguished nodes, then prune Q
- if the nodes and edges of Q' are a subset of nodes and edges of Q, and order(Q') $\leq$ order(Q), and all differing nodes v in (Q – Q') are required and not constrained by values, then prune Q'

We will refer to the above conditions as *eXq* (elimination of query expansions). The essence of *eXq* is to identify not only redundancy of query expansions, but also to respect an induced preference ordering. The last two conditions of eXq guarantee $Q(D) \subset Q'(D)$ for any XML database D: the third condition is the general case of query containment, while the fourth condition requires an additional check for the status of differing nodes of the superquery. This check follows the rationale of the third condition of *eXn*. In the case of a chain of nested queries, i.e., Q1 ⊂ Q2 ⊂ … ⊂ Q(k-1) ⊂ Qk, only Qk may survive the optimization.

The correctness of *eXq* is shown as follows: let us assume that Qi ≡ Qj. Obviously being equivalent Qi will always retrieve the same result set as Qj. For the third condition let us assume that Qi is a superquery of Qj and the differing nodes are not distinguished. Since due to construction of the set of expansions Qi and Qj can only differ in expanded nodes, Qi is more constrained and thus needs witnesses where Qj does not. As these witnesses are not projected in the final result, the result of Qi will always be a subset of Qj. For the fourth condition let us assume that Qi is a superquery of Qj, but the differing nodes are not value-constrained, but required. Then the already shown correctness of *eXn* implies that the result set for Qi is always a subset of Qj.

Consider Q1 and its permuted image in Figure 3(b). We scan the permuted image in reverse order. Each row is iteratively compared with rows higher-up in the permuted image until an equivalence/containment relationship is found. However, if a row is one of equally ranked rows in the preference order, we identify the last row with the same rank in the permuted image and start with that row for a pairwise comparison, instead of simply comparing with higher-up rows. Now row 9 is a potential candidate to be reduced. Since row 9 is a direct subset of row 8 and the differing node (7) is not value-constrained and is a required node in the DTD of Figure 1(a),we can remove row 9 and mark the row with 'X'. Next we find that row 8 is equivalent to row 2 and then remove row 8 because of its lower order. Row 7 is removed because its results are retrieved by row 5. Row 6 is removed because it is equivalent to row 1. We can safely remove row 5, because row 5 is a direct subset of row 2 and the differing node (7) is required in the DTD. By applying the third condition of *eXq*, rows 3 and 4 can finally be removed against rows 1 and 2, respectively. Rows 1 and 2 cannot be further reduced and therefore are marked with an 'O' as the minimal set of expansions. For evaluating the preference query only these two queries have to be posed against the XML database.

The procedure *remove-nestedQuery* in our algorithm *OptiPref* implements the optimization *eXq*. For the purpose of finding a minimal set of order-preserving query expansions, it is sufficient to consider only the unique queries obtained by *eXq* (we omit the proof for reasons of space).

Theorem 1 [Minimality of order-preserving query expansions]: Let Q be a preference twig query, Q1,…, Qn expansions of Q with the minimum size of expanded nodes, and T be any tree DTD. Q1,…, Qk (k ≤ n) are a minimal set of order-preserving expansions of Q, iff there is no nested query Qi (1 ≤ i ≤ k) satisfying *eXq* with respect to some query Qj (1 ≤ j ≤ k), while the order of Qi is lower than the order of Qj.

4.4 Complexity Analysis of *OptiPref*

In this section we analyze the complexity of algorithm *OptiPref* focusing on a key module, *remove-nestedQuery*. The *remove-nestedQuery* module tests nested queries in a permuted image. Since the nesting of a query is repeatedly checked until it is found in the permuted image, it can be done in O(n^2) where n is the number of query expansions. Since the permuted image is created in a sorted fashion, identifying differing nodes of two queries takes linear time. For a pairwise comparison of permuted images, we additionally check all differing nodes v in a super query: if v is a required node, if v is constrained by a value, and if v is a distinguished node. Those can be done by testing *required(*v*)*, *constrained(*v*)*, and *projected(*v*)*, respectively, which can be organized in a hash table for efficient execution. Thus, it takes O(n^2 * m * e), where n is the number of query expansions, m is the query size, and e is the number of expanded nodes in the query. The complexity of identifying nested queries is O(n^4). As a result, the complexity of algorithm *OptiPref* is O(n^4) in the size of a set of expansions.

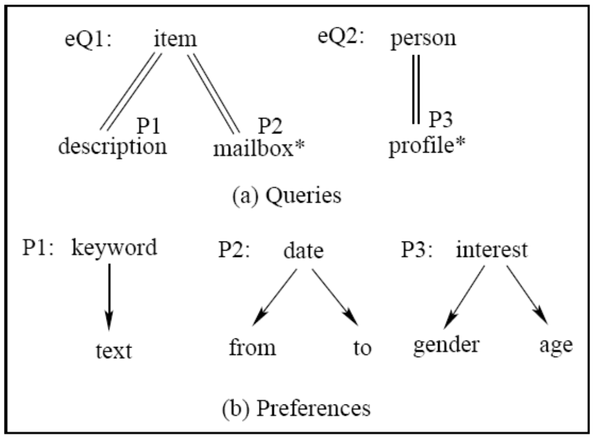


Figure 5: Queries and preference constraints

5. EXPERIMANTAL RESULTS

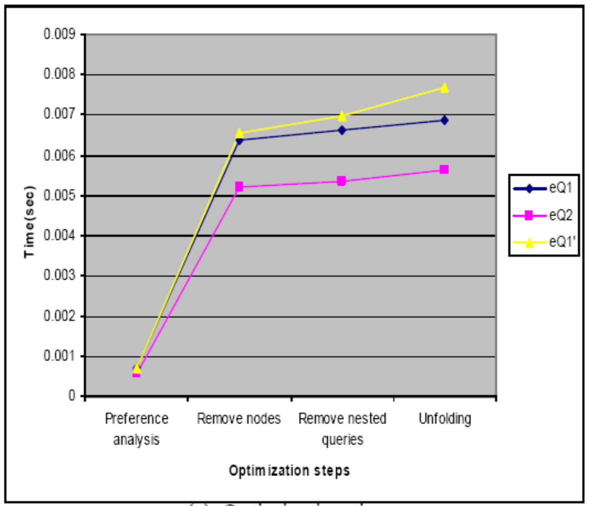
5.1 Experimental Setup

We ran extensive experiments using the *XMark* benchmark dataset. The DTD of XMark is given at http://monetdb.cwi.nl/xml/. For all experiments, we used that all preference attributes were required in the DTD, if they were valid. Our experiments were run on SUN UltraSPARC-III workstation with 750 MHZ CPU and 8,192 MBytes of main memory. In all experiments, time is reported in seconds. As for query evaluation we used a structural join algorithm for XML databases [8].

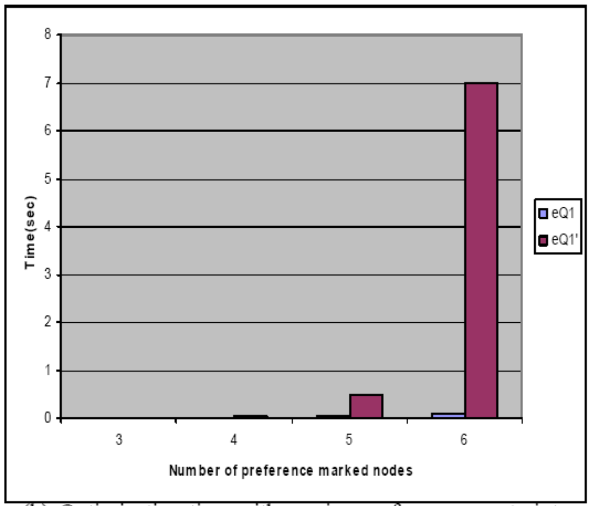
Figure 5(a) and (b) show the queries and structural preference constraints we used in our experiments. The query eQ1 is a branch query, where the nodes *description* and *mailbox* are marked with P1 and P2 preferences, respectively, whereas the query eQ2 is a simple path query, where its leaf node *profile* is marked with a P3 preference constraint. Additionally we constructed eQ1' having the same structure as eQ1, but considered all leaves as distinguished. All attributes in preferences P1, P2, and P3 are valid in the DTD.

5.2 Studying Optimization Time

In order to measure optimization times, we ran three sets of experiments. In the first set we study the optimization time needed for each optimization steps. We ran the three queries eQ1, eQ2, and eQ1'. Figure 6(a) shows the cumulative optimization times of the queries. The query eQ1' took a bit more time than the query eQ1, because eQ1' has more surviving queries after optimization. Thus, it took more time to unfold the query. The query eQ2 took



(a) Optimization times



(b) Optimization time with varying preference constraints

Figure 6: Studying optimization time

less time than the queries eQ1 and eQ1', because its expansions are smaller. The graph shows that the time spent in removing expanded nodes is approximately 50% of the total optimization time. This is due to the routine also including the time for creating the permuted image. The time to identify nested queries are only about 4% of the optimization time. However, when increasing the number of preferences in the query, the time for determining nested queries also increases rapidly. For example, when we increase 3, 4, 5, and 6 preference constraints in eQ1, the routine of testing nested queries takes 23%, 68%, 95%, and 99% of the optimization times, respectively.

In the second set we varied the number of preferences in the queries from 3 to 6. We constructed the queries eQ1 and eQ1' by increasing the query size with preference constraints, by which all leaf nodes in both queries are marked. Whenever a new leaf node is added, it is marked with a different preference of size 3. One difference is that eQ1 maintains only a single distinguished node, while eQ1' treats all leaves as distinguished nodes. The sizes of permuted images for eQ1 and eQ1' are 48, 192, 768, and 3072, when 3, 4, 5, and 6 query nodes are labeled with preference constraints. The graph of Figure 6(b) shows the variation in optimization time. The optimization time of eQ1 is from 1.4 up to 78 times lower than that of eQ1', because more expansion queries can be pruned in eQ1. Due to its polynomial complexity the optimization time increases with a growing number of query nodes annotated with preference constraints.

The third set of experiment used the three queries and varied the size of preferences from 10 to 50 while keeping the original query size fixed (3 nodes). The sizes of permuted images for eQ1/eQ1' are 11^2, 21^2, 31^2, 41^2, and 51^2, when the sizes of preference constraints vary from 10, 20, 30, 40, to 50, respectively (the sizes of permuted images for eQ2 are 11, 21, 31, 41, and 51). The graph of Figure 7(a) shows that the optimization times of the queries eQ1 and eQ2 increase slightly for different sizes of preferences, whereas the optimization time of query eQ1' dramatically increases, for the same reason as given in the second experiment also holds for this case.

In summary, the optimization time depends on the number and the size of preference constraints. Moreover, in absolute numbers we can see that the overhead of optimization time is very small compared to query evaluation times, which we will study next.

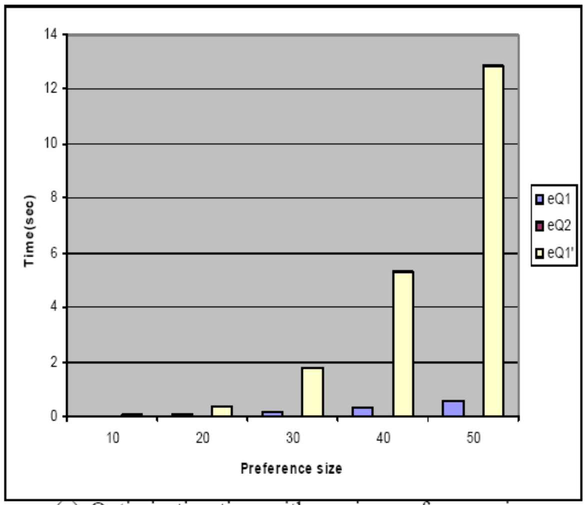
5.3 Studying Evaluation Time

This experiment shows the benefits of our optimization algorithm. We first ran queries eQ1, eQ2, and eQ1' and varied the size of dataset from 50MB to 100MB. We measured evaluation times of optimized and not optimized queries.

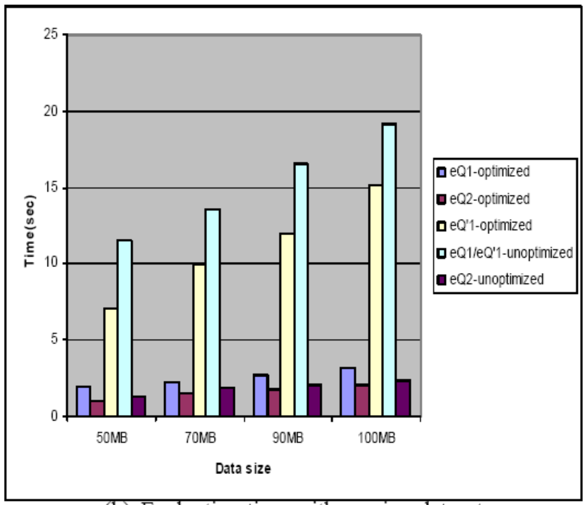
The numbers of optimized expansions of eQ1, eQ1', and eQ2 are 3, 6, and 3, respectively. The numbers of not optimized expansions of eQ1/eQ1'and eQ2 are 12 and 4, respectively. Figure 7(b) reports query evaluation times of optimized and not optimized queries eQ1, eQ2, and eQ1' with varying dataset sizes. The total evaluation time is obtained by adding up evaluation times of all query expansions. The optimized queries eQ1 and eQ2 are about 6 and 1.2 times cheaper than the not optimized ones, respectively, because eQ1 has larger benefit from our optimizations, but eQ2 does not. The optimized query eQ1' is about 4.5 times more expensive than the optimized query eQ1 due to the size of optimized expansions. The not optimized query eQ1 results in far higher overall evaluation times.

In order to understand the optimized evaluation better, we ran two additional sets of experiments. In the first set, we constructed queries by adding query nodes with preference constraints. In the second set, we build more complex preference constraints by increasing the number of preferred attributes.

Varying the number of preference constraints: We measured the evaluation time of the second set of our experiment in Section 5.2 over a 50MB dataset. We varied the number of preferences from 2 to 4 in the queries, where the size of a new preference constraint is 2. The number of optimized and not optimized expansions of eQ1 and eQ1' queries with varying numbers of preference constraints is given in Table 1.



(a) Optimization time with varying preference size



(b) Evaluation time with varying dataset

Figure 7: Optimization and evaluation times

Number of preferences	2	3	4
Optimized eQ_1	3	3	3
Optimized eQ'_1	6	18	54
Unoptimized eQ_1/eQ'_1	12	48	192

Table 1: Number of query expansions

The graphs of Figure 8(a) report times for three cases: the optimized eQ1, optimized eQ1', and not optimized eQ1/eQ1'. The time to perform optimized evaluation for both eQ1 and eQ1' is significantly smaller than that in non-optimized evaluation with growing number of preference constraints. Regarding evaluation times, especially for eQ1 the benefits of our optimization are substantial: while increasing the number of preferences the evaluation time for the optimized eQ1 slightly increases. This is because our optimization algorithm identifies that many expansions containing redundant preferred attributes can be dropped. For example, with 4 preference constraints in the queries, the optimized evaluations of eQ1 and eQ1' have savings of 90% and 50% over the non-optimized case, respectively. The evaluation of the optimized eQ1 is approximately 4 times cheaper than that of the optimized eQ1'.

The larger the number of query nodes marked with preferences, the more time is needed for non-optimized queries. Actually, this increase appears to be exponential in the number of query nodes marked with a preference constraint. The time difference between optimized and non-optimized evaluations heavily grows with an increasing number preference constraint.

Varying the size of preference constraints: in this experiment, we measured the evaluation times with the third set of experiments in Section 5.2 and again used a 50MB dataset. The number of optimized and non-optimized expansions of eQ1, eQ2, and eQ1' for varying sizes of the preference constraints is given in Table 2. Figure 8(b) shows that for a fixed query size (3 nodes) and a growing size of preference constraints (from 4 to 7 nodes), the evaluation times of the optimized eQ1 and eQ1' are 17, respectively 1.5 times cheaper than that of any of the non-optimized queries eQ1/eQ1'. Additionally, the evaluation time of the optimized eQ2 is 1.2 times cheaper than that of non-optimized eQ2. The graph shows that evaluation times grow in a linear fashion for growing sizes of the preference constraints. Especially the evaluation time for the optimized eQ1 only increases slightly.

Size of preference	4	5	6	7
Optimized eQ_1	4	5	6	7
Optimized eQ_2	4	5	6	7
Optimized eQ'_1	16	25	36	49
Unoptimized eQ_1/eQ'_1	25	36	49	64
Unoptimized eQ_2	5	6	7	8

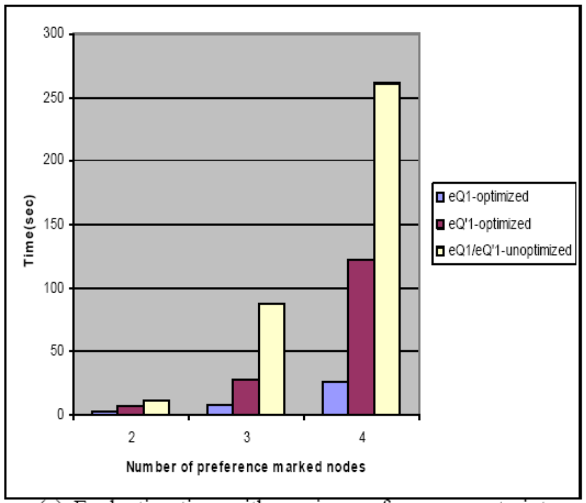
Table 2: Number of query expansions

The benefits of our optimization techniques are similar in all experiments. In particular, our experiments show that our optimization decreases the number of necessary expansion queries up to two orders of magnitude and thus also evaluation times are substantially reduced. Dealing with structural preferences the naive unfolding encounters a combinatorial explosion resulting in a heavy replication of query parts retrieving common subsets. The optimized unfoldings, however, are substantially less affected by growing numbers and sizes of preference constraints.

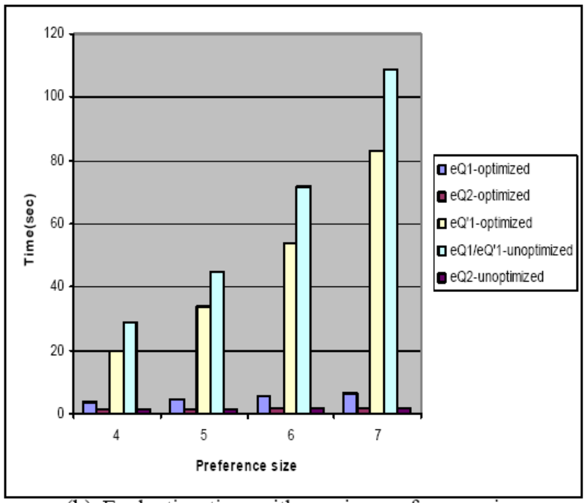
6. RELATED WORK

Due to the importance in practical applications the optimization of XPath and XQuery queries has attracted considerable attention leading to a large number of possible techniques (see, e.g., [9-11, 7, 12, 13]). In particular, [10] focuses on node minimization to obtain equivalent queries of smallest size for subsequent execution, and [12] stressed the benefits of query rewriting in XML databases using DTD information.

Since an exact match retrieval model is often inadequate in practical scenarios, XML query engines have already been extended to support an IR-style matching of data values. Engines like XIRQL [14] and XXL [15] applied probability-based keyword retrieval to



(a) Evaluation time with varying preference constraint



(b) Evaluation time with varying preference size

Figure 8: Studying evaluation time

structured XML documents. Though these engines allow finding similar values for a certain predicate or relaxing query predicates to find desired values in related structural elements, they do not support real categorical user preferences like handled by our approach.

Several systems for supporting preference queries have been recently proposed. In [16], the author investigated semantic optimization of preference queries to remove redundant occurrences of preferred values in relational databases. Recent work by [6, 17] proposed preference XML queries in connection with the modeling framework for preferences as partial order graphs given in [3]. In particular, in [6], the authors proposed PreferenceXPath by extending XML queries using the XPath standard. The resulting language enables the use of soft filtering conditions in contrast to conventional exact match conditions for node-selection in XPath. Like in our approach a soft condition defines a strict partial order over the set of elements to be filtered and then returns only the best matches. For optimization, however, PreferenceXPath queries have to be mapped to relational algebra expressions (extended by constructs like preference-selections) for choosing execution plans using hill climbing optimization [17].

To our knowledge none of these approaches dealt with structural preferences in XML queries. Building on previous approaches for preferences in information systems, in this paper we presented a framework of how twig queries can be expanded by such preference information. Queries have to be unfolded into a set of queries that is guaranteed to retrieve every possibly relevant document from an XML database. However, if more than a single preference is involved, the unfolding of a query leads to a set of queries, where some individual queries may retrieve common subsets, or even exactly the same data. Hence, our focus is on optimizing the necessary set of queries while preserving the induced order of the result set.

Another related area of work is presentational preferences for XML query results. Although in contrast to our approach these preferences do not affect the matching process, some basic techniques exploiting the DTD structure are related. Since XQuery is a data-transformation query language, users can easily define a set of preferences to impose an ordering on the presentation of the results. The work in [9] focused on a rewriting technique for queries on XML data streams that during the processing of an XML document can output results respecting a user's presentational preferences. This rewriting uses the structure of the DTD to first perform a set of algebraic optimizations to queries and then also exploits order constraints in the DTD to schedule the evaluation of subexpressions of the query. However, all optimizations focus on minimizing the use of main memory buffers, rather than execution speed.

7. CONCLUSIONS

In this paper we have considered the problem of order-preserving optimization of twig queries with structural user preferences given a DTD. Our technical contributions are summarized as follows.

- First we showed how to expand twig queries with structural elements provided by a user's preferences. Here, performing a simple check against the DTD always guarantees that queries are only expanded by relevant elements.
- We investigated novel optimization techniques that allow to remove all redundant expanded nodes from individual queries, and eliminate nested queries from the unfolded set, thus identifying a minimal set of query expansions.
- We devised a complete and efficient polynomial time algorithm, OptiPref, which took worst-case time $O(n^4)$ in the size of a set of expansion queries.
- Our experimental results have shown the essential benefits of our algorithm in term of query evaluation times for varying complexities of preference constraints and changing numbers of preferences in a query. As can be expected the benefits generally are

dramatically growing with more complex preferences and more nodes in the query that contain preferences.

Moreover, we analytically showed that our algorithm always obtains the minimal set of expansion queries, where each query contains only a minimum of expanded nodes given by preferred elements, while the natural order induced by the preferences is always preserved. It is also important to point out that our optimization techniques do never alter the result set as given by the unoptimized set of unfolded queries.

Our future work will address the incorporation of a scoring method that subsequently retrieves best matching answers with score information. Moreover, we will explore the integration of our framework into existing preference frameworks for practical applications.

8. ACKNOWLEDGMENTS

This work was funded within the Emmy-Noether Program of Excellence by the German Research Foundation (DFG).

9. REFERENCES

[1] Chomicki, J.: Querying with intrinsic preferences. In: Proceedings of EDBT. (2002)

[2] Agrawal, R., Wimmers, E.: A framework for expressing and combining preferences. In: Proceedings of SIGMOD. (2000)

[3] Kießling, W.: Foundations of preferences in database systems. In: Proceedings of VLDB. (2002)

[4] Stolze, M., Rjaibi,W.: Towards scalable scoring for preference-based item recommendation. In: Bulletin of the IEEE Technical Committee on Data Engineering. (2001)

[5] Balke, W.T., Wagner, M.: Through different eyes assessing multiple conceptual views for querying Web services. In: Proceedings of WWW Conf. (2004)

[6] Kießling,W., Hafenrichter, B., Fischer, S., Holland, S.: Preference XPATH: a query language for E-commerce. In: Proceedings of Wirtschaftsinformatik. (2001)

[7] Miklau, G., Suciu, D.: Containment and equivalence for an XPath fragment. In: Proceedings of PODS. (2002)

[8] Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J., Srivastava, D., Wu, Y.: Structural joins: efficient matching of XML query pattern. In: Proceedings of ICDE. (2002)

[9] Koch, C., Scherzinger, S., Schweikardt, N., Stegmaier, B.: FluXQuery: an optimizing XQuery processor for streaming XML data. In: Proceedings of SIGMOD. (2004)

[10] Amer-Yahia, S., Cho, S., Lakshmanan, L., Sirvastava, D.: Minimization of tree pattern queries. In: Proceedings of SIGMOD. (2002)

[11] McHugh, J., Widom, J.: Query optimization for XML. In: Proceedings of VLDB. (2000)

[12] Wood, P.: Containment for XPath fragments under DTD constraints. In: Proceedings of ICDT. (2003)

[13] Papakonstantinou, Y., Vassalos, V.: Query rewriting for semi-structured data. In: Proceedings of SIGMOD. (1999)

[14] Fuhr, N., Großjohann, K.: XIRQL: A query language for information retrieval in XML Documents. In: Proceedings of SIGIR. (2001)

[15] Theobald, A., Weikum, G.: The index-based XXL search engine for querying XML Data with relevance ranking. In: Proceedings of EDBT. (2002)

[16] Chomicki, J.: Semantic optimization of preference queries. In: Proceedings of International Symposium on Applications of Constraint Databases. (2004)

[17] Hafenrichter, B., Kießling, W.: Optimization of relational preference queries. In: Proceedings of Australasian Database Conference (ADC). (2005)