

one. Again, these works are orthogonal to our approach since their objective is different. In summary, to the best of our knowledge, only [9] explicitly focuses on XPath expressions, and this in the context of query containment without considering how to derive useful information from a DTD. All other approaches either propose physical (cost-based) query optimization techniques or assume data models and query languages (typically regular path queries) that are not fully comparable to XPath expressions.

Structure of the Paper. After some background information on XPath and DTDs in Section 2, we present the concept of path dependencies and the computation of path equivalence classes in Sections 3 and 4. Our optimization scheme for XPath expressions is detailed in Section 5. Section 6 presents an evaluation we conducted for our approach.

2 XPath and DTDs

XPath assumes that XML documents are modeled as rooted, node-labeled trees. Let Σ be a set of element names. An XML document d has a set V_d of nodes with $v_r \in V_d$ being the root node. With each node $v \in V_d$ an element name (label) $e \in \Sigma$ is associated. Each node has a (possibly empty) list of child nodes. Finally, every element $e \in \Sigma$ can have zero or more element attributes. With each node $v \in V_d$, a *path* can be associated, which consists of a sequence of element names of nodes, starting with label of the root node v_r and ending in the label of v . Obviously, different nodes in a document can have the same path.

Given an XML document d , an XPath expression evaluates to a set of nodes in d such that their properties “match” the pattern(s) specified in the expression (in terms of paths, children etc). A regular XPath expression is based on the following grammar (using the abbreviated XPath syntax). Let $e \in \Sigma$ be an element name, a an element attribute, and s a string.

regular path := $p_1 \mid p_1/p_2 \mid /p_1 \mid //p_1 \mid \cdot \mid \dots \mid p_1/p_2 \mid p_1//p_2 \mid p_1[predicate] \mid ancestor\text{-or-self} \mid e \mid * \mid @a \mid @* \mid text() \mid node()$

predicate := $q_1 \text{ and } q_2 \mid q_1 \text{ or } q_2 \mid \text{not } q_1 \mid (q_1) \mid p \mid p = s$

We assume that the reader has some familiarity with XPath [6] and the semantics of XPath expressions [20]. We thus only outline the major characteristics of regular paths that are important to our approach. In general, one distinguishes between an *absolute path* that starts from the root node of the document and a *relative path* that starts from the context node (“.”), which can be determined outside the single path pattern (e.g., a subquery in an XQuery expression). A path pattern is a sequence of one or more *location steps*. A location step is applied to a context node and consists of (1) an axis, which specifies the tree relationship between the nodes to be selected by the location step and the context node, (2) a node test specifying

the type of nodes to be selected by the location step, and (3) zero or more predicates, which further refine the nodes to be selected in the location step.

In a location step, an axis specifies the relationship between the context node and the nodes to be selected next. There are 13 axes of which the child axis “/” (which is also the default axis), descendant[-or-self] axis “//” (assuming document order is irrelevant), ancestor[-or-self] axis, and parent axis “..” are relevant for our framework. The other axes deal with attributes and namespaces, and relationships that explicitly refer to the document order in which the nodes to be selected occur. As done in other works, e.g., [1, 2], we currently take a *data centric view* (compared to a *document centric view*) of XML documents where the order of sibling nodes is not relevant. Most important for our approach, for a context node, a predicate filters a node-set with respect to one or more axes to produce a new node-set. In the above grammar, a predicate in a regular path is enclosed in “[]”. Predicates can be combined through the logical connectives “and” and “or” and can also be nested since a predicate can again contain a regular path. Given an XPath expression, we denote the regular path that is obtained by removing all predicates (i.e., all “[]” components) as *selection path*. All regular path expressions that occur in a (nested) qualifier of an XPath expression are called *condition paths* or simply conditions.

Example 2 Recall the XPath expression from Ex. 1. The selection path is `//buyer/person/name` and the two conditions are `[email | phone]` and `[annotation[happiness]/author = "John"]`. The corresponding regular paths for the conditions are `//buyer/person/email`, `//buyer/person/phone`, `//auction/annotation/author` and, `//auction/annotation/happiness`. □

We use Document Type Definitions (DTDs) as the schema formalism for our approach. A DTD basically specifies admissible elements, element nesting, and element attributes in form of an extended context-free grammar. For this, with each element a *content model* in form of a regular expression is associated. Because of space limitations, in the following, we only consider non-recursive DTDs. A discussion of our framework including recursive DTDs can be found in [15].

3 Path Equivalence Classes

We now introduce the concepts of path dependency and path equivalence classes (PECs), which aggregate schema information in the DTD in a formal and concise way. The computation of PECs from a DTD is detailed in Section 4.

Let d be an XML document that conforms to a DTD D . The *path extent* of a path p , denoted $extent_d(p)$, is defined as the set of all nodes in the document d having p as path. The following relationship is established among nodes that are members of such extents.

Definition 1 (Descendant Nodes) Let p_1, p_2 be two paths in d . $desc_d(p_1, p_2)$ denotes the subset $V' \subseteq V_d$ of nodes in d such that every node $v \in V'$ (1) is a member of $extent_d(p_1)$ and (2) has a descendant node $v' \in V_d$ with v' being a member of $extent_d(p_2)$. \square

This definition captures the aspect that with all nodes having the same path, another set of descendant nodes having a different path co-occurs. Let $LCP(p_1, p_2)$ denote the longest common prefix of two paths p_1 and p_2 . In the following, we introduce some important notions that form the basis for path equivalence classes and relationships among such classes.

Definition 2 (Path Dependency, Implication and Contradiction) Let p_1 and p_2 be two paths and let $l = LCP(p_1, p_2)$. We define

- 1) p_1 implies p_2 , denoted $p_1 \rightarrow p_2$, if and only if $desc_d(l, p_1) \subseteq desc_d(l, p_2)$,
 - 2) p_1 and p_2 are path-dependent, denoted $p_1 \sim p_2$, if and only if $desc_d(l, p_1) = desc_d(l, p_2)$, and
 - 3) p_1 and p_2 are contradictory, denoted $p_1 \perp p_2$, if and only if $desc_d(l, p_1) \cap desc_d(l, p_2) = \emptyset$,
- for any document d that conforms to a DTD D . \square

Theorem 1 The path dependency relation \sim is an equivalence relation. \blacksquare

Definition 3 (Path Equivalence Classes, PECs) The set of equivalence classes defined by the relation \sim is called path equivalence classes (PECs). For a path p , $[p]$ denotes the path equivalence class to which p belongs. \square

Suppose an XPath expression whose result relies on the existence of several condition paths. Intuitively, if all these paths belong to the same PEC, then they are all path-dependent and we can conclude that only one of those is necessary and the others are redundant. Next, we will introduce some properties of PECs.

Definition 4 (PEC Representative) Let C be a PEC. A path $p \in C$ is the PEC representative of C , denoted $rep(C)$, if and only if $|p| \leq |p_i|$ for all $p_i \in C$, i.e., p is the shortest path in C . \square

If a query includes a condition path $p \in C$, one can always replace p with $rep(C)$ because of the path dependency property of paths in C . We choose the shortest path as representative since for query evaluation a shorter path generally means less graph traversal.

Theorem 2 Let p and q be two paths in a document d . Suppose $p' \in [p]$. Then the following holds:

- 1) if p implies q , then p' implies q ,
- 2) if p and q are contradictory, then p' and q are contradictory. \blacksquare

In other words, if a path p implies (contradicts) another path q , so do all other paths that belong to the same class as p . The proof is straight-forward. Based on the sharing of properties among paths in a PEC,

one can establish relationships between two paths p and q , and respective PECs.

Definition 5 (PEC Implication and Contradiction) Let $[p]$ and $[q]$ be two PECs. $[p]$ implies $[q]$, if and only if p implies q , for any document d conforming to the DTD D . Analogously, $[p]$ contradicts $[q]$, if and only if p and q are contradictory, for any document d conforming to the DTD D . \square

Having two condition paths in different classes does not always mean that they are necessary conditions. If there is an implication between the classes, then one of them is implied by the other and thus can be eliminated. On the other hand, if the classes are found to be contradictory, logically connecting two conditions by an “and” operator always results in false.

4 Computing PECs and Relationships

We now detail a three step approach to compute and aggregate information about PECs from a given DTD. First, individual content models for elements in the DTD are investigated (Section 4.1). Then, a so-called *X*Trie is computed that captures admissible paths according to the DTD (Section 4.2). Finally, information in an *X*Trie is aggregated in a *Class Implication and Contradiction Graph* (Section 4.3).

4.1 DTD Preprocessing

In the DTD preprocessing step, the content model for each element in a given DTD D is investigated. Information about what elements are required/optional for a given element and how the child elements are related is recorded in an *Element Relationship Table (ERT)*.

Assume a content model m_e for element $e \in \Sigma$. Based on the specification of e 's content model in D , child elements can be optional, required, and/or can occur in a choice group. An element e' is said to be a required element in the content model of e iff for every document that conforms to the given DTD, there exists at least one child element e' for every occurrence of e . Otherwise, element e' is said to be optional. The other type of information in an ERT describes the relationships among the elements in e 's content model. Assume a content model for e where e' and e'' are siblings. e' implies e'' , denoted $e' \rightarrow e''$ iff for every document that conforms to D and for every occurrence of e' , there exists a sibling element e'' for e' . e' contradicts e'' if iff for every document and for every occurrence of e , there exists no sibling e'' for e' . e' and e'' are dependent, denoted $e' \sim e''$, if imply each other. The following example lists some entries of the ERT for the DTD shown in Example 1. (r) and (o) stand for required and optional, respectively.

Element	Children	Dependencies
<i>auctions</i> <i>person</i>	<i>auction(o)</i> <i>name(r), email(r),</i> <i>phone(o)</i>	<i>email ~ name,</i> <i>phone → name,</i> <i>phone → email</i>
<i>item</i>	<i>name(r), descr(o),</i> <i>link(o)</i>	<i>descr ~ link,</i> <i>descr → name,</i> <i>link → name</i>

The computation of an ERT occurs when a DTD is validated at schema definition time. Obviously, if a DTD has n elements, then the ERT has n entries. For a content model $m_i, i = 1, \dots, n$ with k_{m_i} elements specified in m_i , at most $k_{m_i}^2$ comparisons are necessary to determine the relationships among all k_{m_i} . Let $k = \max\{k_{m_i}^2 \mid i = 1, \dots, n\}$. Then the time complexity for constructing an ERT is bound by $O(n * k)$.

4.2 XTriE

The next step in preparing structures for optimizing XPath expression is to model admissible paths for a given DTD D . For this, we choose a relatively simple derivative of a DTD, called *XTriE*, which enumerates all possible paths in XML documents that conform to D . An *XTriE* is a simplified schema formalism in the sense that information about alternative content models, groupings, and repetitions is not modeled.

Definition 6 (XTriE) Let Σ be a set of element names. An *XTriE* is a 6-tuple $\langle S, s_0, E, slabel, elabel, etype \rangle$, where

- S is a set of states with a distinguished start state, s_0 . We use S^* to denote the set $S - s_0$.
- $E \in S \times S$ is a set of edges.
- *slabel* (state label) is a mapping from S to $\Sigma \times \mathbb{N}$, and *elabel* (edge label) is a mapping from E to Σ .
- *etype* (edge type) is a mapping from E to $\{\text{optional}, \text{required}\}$. \square

Except for the start state s_0 , every state $s \in S$ in an *XTriE* corresponds to exactly one possible path in any XML document that conforms to the DTD from which the *XTriE* has been derived. Labels for states and edges are given by the mapping *slabel* and *elabel*, respectively. An edge between two states (e, i) and (e', j) is of type *required* if element e has element e' as a required child in its content model. Otherwise, an edge is of type *optional*. Figure 1 shows the *XTriE* derived from the DTD given in Example 1. For example, the path *auctions.auction.payment.creditcard* results in the state $(\text{creditcard}, 1)$. The same element can occur in different states, depending on how it is used in content models (e.g., elements *name* and *phone*). The meaning of the states, based on by what types of edges and edge labels they are reached, is obvious.

We now give the algorithm for deriving an *XTriE* from a DTD D . We first initialize an agenda for states and edges to be created:

1. Let Σ be the set of element names that occur in D . For each $e \in \Sigma$, initialize a counter $c_e := 1$
2. For the document root element with label r , add a state labeled $(r, 1)$ to S .
3. $S := S \cup \{(s_0, 1)\}$; add an edge of type *required* from $(s_0, 1)$ to $(r, 1)$ labeled r to E
4. Initialize an agenda list *Agenda* containing $(r, 1)$.

while *Agenda* is not empty **do**
 remove a state, say (e, i) , from *Agenda*
for each element t occurring in e 's content model **do**
 $S = S \cup (t, c_t)$; $c_t = c_t + 1$;
 add an edge β from (e, i) to (t, c_t) with label t to E
if t is a required child
then $etype(\beta) = \text{required}$ **else** $etype(\beta) = \text{optional}$
 add (t, c_t) to *Agenda*

Conceptually, an *XTriE* can be computed directly from the DTD. However, since it is possible for an element to appear in different content models, one may need to check an element multiple times to see whether it is a required or optional element in the current context. In order to avoid parsing and analyzing a possibly complicated content model more than once, we simply query the ERT constructed previously from the DTD. The ERT thus can be considered as an optimization technique to speed up the construction. In the above algorithm, information from the ERT is requested for checking the **if** condition.

Given a DTD with n elements, the depth of an *XTriE* is less than or equal to n for a non-recursive DTD. For a state at depth l , there are at most $n - l$ outgoing edges. Thus, the number of states in an *XTriE* is bound by $O(2^n)$. In practice, however, the size of an *XTriE* and its computation is much less, as we will show in Section 6. Assuming that information about elements in the ERT can be accessed in constant time (e.g., using a hash table), the total number of items to be considered in the *Agenda* is bound by the number of states in the *XTriE*.

4.3 CIC-Graph

The final step is to aggregate information about paths and path equivalence classes into a compact structure that can efficiently be used to optimize XPath expressions at query compile time. The basic idea here is that in an *XTriE*, each state different from the start state represents exactly one admissible path. Consequently, finding all PECs for a DTD corresponds to partitioning the set of states S^* in an *XTriE*, with each partition representing a PEC (see the regions in Fig. 1).

In the following, we detail how such a so-called *Class Implication and Contradiction graph* (CIC-graph) can be efficiently constructed from an *XTriE*. PECs are abstracted as nodes in the CIC-graph. Each node in the graph is linked to a set of states S_p in the *XTriE* such that S_p constitutes a PEC. The relationships among

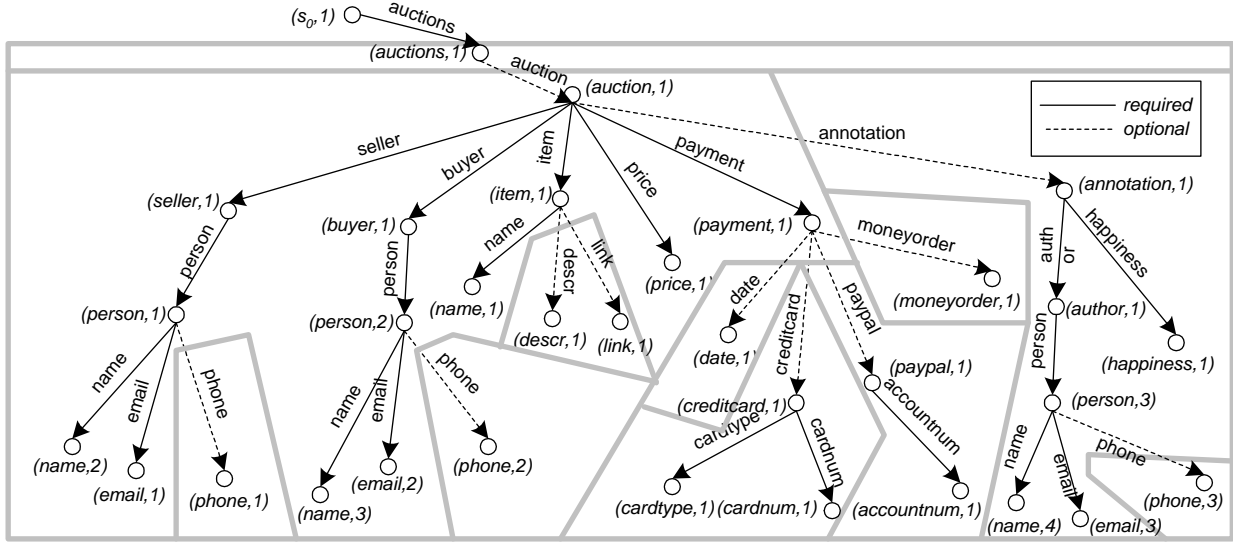


Figure 1: XTrie for the Auction DTD in Example 1

PECs are represented in the graph by two types of edges: i-edges (for implication) and c-edges (for contradiction). With an XTrie and CIC-graph as the supporting structures, one can efficiently determine the PEC to which a path belongs and the existence of implication or contradiction between any two paths; these are crucial operations in optimizing an XPath expression as discussed in Section 5.

Let X_D be the XTrie constructed from a DTD D . A CIC-graph, $H = (V, E_i, E_c, link)$, is a directed graph with a set V of nodes, and two sets of edges, E_i and E_c , representing implication and contradiction, respectively. $link$ is a mapping from S^* , the set of states in X_D , to V . The construction of a CIC-graph is a two step process, (1) state collapsing, and (2) edge insertion and node merging.

1) State collapsing: Assume that optional edges are removed from the XTrie. From the transitivity property of path dependency, it is easy to see that the states within a connected subtree belong to the same PEC. Thus, the first step in constructing H is to collapse the set of states in each subtree t containing the set of states S_t connected by required edges in X_D into a single node n . We then insert each n into V , and insert (s, n) to $link$ for each $s \in S_t$. Next, we discuss how to transform the edges remaining in the collapsed XTrie to an initial set of edges for H .

There are two types of path implication. One is induced by the ancestor-descendant relationship, and the other is induced by the sequence group construct in a content model. For the former, since the edges in the collapsed XTrie represent parent-child relationships and we know that any descendant implies his ancestor, we can simply reverse the direction of these edges and use them as the initial set of i-edges for H .

That is, for each (e_1, e_2) in the collapsed XTrie, we insert $(link(e_2), link(e_1))$ to E_i .

2) Edge insertion and Node Merging The graph H is further enriched by the additional information in the ERT regarding sibling relationships. i-edges and c-edges are inserted into H accordingly using the rules in the ERT. Two nodes in H are merged if the states linked to the nodes are found to be dependent.

Since required elements have already been considered in step 1), we now focus on optional elements only. For each XTrie state $x \in S$ with k optional children, $k > 1$, let $Q_1, \dots, Q_k \in V$ be the nodes in H the optional children $c_1, \dots, c_k \in S$ link to, and let R_x be the set of rules from the ERT for the element x . For each rule $r_i \in R_x$ containing elements c_{j_1} and c_{j_2} , where $1 \leq j_1, j_2 \leq k$, if r_i is an implication rule, $c_{j_1} \rightarrow c_{j_2}$, we insert an i-edge (Q_{j_1}, Q_{j_2}) to E_i . Similarly, if r_i specifies a contradiction between c_{j_1} and c_{j_2} , two c-edges (Q_{j_1}, Q_{j_2}) and (Q_{j_2}, Q_{j_1}) are inserted to E_c as contradiction is a symmetric relation. For a dependency between c_{j_1} and c_{j_2} , Q_{j_1} and Q_{j_2} are merged into one node. This is done by removing Q_{j_2} from V and replacing all occurrences of Q_{j_2} in E_i , E_c and $link$ by Q_{j_1} .

The CIC-graph for our example DTD is shown in Fig. 2. As an example, the i-edge from *paypal* to *date*, and the c-edges between *paypal* and *moneyorder* are the results of applying the dependencies in the *payment* entry of the ERT. Furthermore, nodes *descr* and *link* are merged because they are dependent.

Now, each path equivalence class for the DTD is represented by exactly one node in the graph. Therefore, two paths are in the same class (path-dependent) if and only if their matching XTrie states link to the same node in the CIC-

graph. For example, `auctions.auction.item.descr` and `auctions.auction.item.link` are path-dependent, whereas `auctions.auction` and `auctions.auction.annotation` are not.

Using the transitivity property of implication, one can easily identify PEC implications and contradictions using the CIC-graph. For two classes C_1 and C_2 , one can determine whether C_1 implies C_2 by checking if there exists a sequence of i-edges from C_1 to C_2 . Similarly, C_1 contradicts C_2 if there exists a c-edge between them, or between the classes they imply. For example, the path `auctions` is implied by `auctions.auction.annotation.phone`, though they are not directly connected.

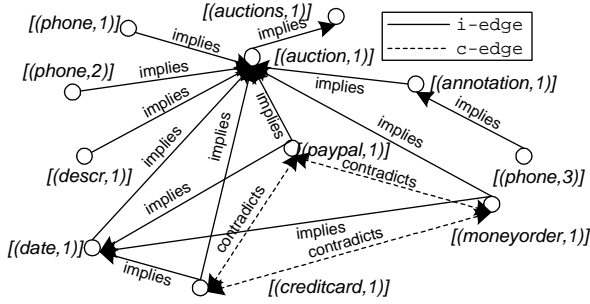


Figure 2: CIC-Graph for the Xtrie in Figure 1

Let $n = |S^*|$ be the number of nodes in an Xtrie, excluding the start state. In the worst case, each PEC contains exactly one path, and is implied by or contradicts every other PEC. This results in n nodes and $n^2 - n$ edges in the CIC-graph. The size of the graph is then bound by $O(n^2)$. Step 1) of the algorithm requires one scan of the Xtrie, which can be done in time $O(n)$. For each pass of the loop in Step 2), we either generate an edge or merge two nodes. Since there are $O(n^2)$ edges in the graph, Step 2) requires time $O(n^2)$. Therefore, the construction time for a CIC-graph is bound by $O(n^2)$.

5 Optimization

PECs and their relationships computed from a DTD at schema-compile time are now used to optimize XPath expressions before they are passed to a query evaluation engine. After an overview of this optimization process in Section 5.1, we first focus on the decomposition of an XPath expression in Section 5.2, and then detail the core optimization step in Section 5.3. Finally, in Section 5.4, we outline how resulting optimized XPath expressions are reconstructed.

5.1 Overview

Given a DTD D and its corresponding Xtrie and CIC-graph, respectively, we now detail an algorithmic framework used to rewrite an XPath expression p into an XPath expression p' such that (1) p and p'

are semantically equivalent, and (2) p' is an optimal version of p . Semantically equivalent means that for any document that conforms to D , p and p' evaluate to the same node-set. The notion of optimality is slightly harder to define. Ideally, it should capture the number of physical disk or block accesses necessary to evaluate the expression. Since our optimization approach operates on the logical and not the physical level, we have to utilize a slightly weaker notion.

Definition 7 (Optimal XPath Expression) Given an XPath expression p . An XPath expression p' is said to be optimal with respect to p iff p and p' are semantically equivalent, (1) p' does not contain redundant conditions, and (2) no condition or selection path in p' can be shortened. \square

The optimization process consists of three phases, (1) decomposition, (2) optimization, and (3) reconstruction. During decomposition, an XPath expression p is translated into an XPath Condition Tree (XCT), which is an AND-OR tree that describes the logical connectivity among the selection and condition paths in the expression. In the second phase, wildcards in all paths of the expression are expanded. After this, rewrite rules based on PECs are applied to obtain an optimized combination of condition and selection path(s). Finally, in phase 3 an optimized XPath expression p' is reconstructed from the XCT.

5.2 Decomposition and XCT Construction

For decomposition, first the different components of the input XPath expression are indexed using subscript marking. Then the expression is translated into an XPath Condition Tree.

Subscript Marking. As discussed in Section 2, an XPath expression can contain multiple, possibly nested predicates (condition paths). The positions of these predicates in the expression play an important role in query evaluation. Subscript marking is a mechanism to memorize the position of each condition path so that a condition that is not redundant can be placed in the correct position when the optimized expression is reconstructed. For this, each node test in the expression is marked with a subscript $c \in \mathbb{N}^*$. The following example shows some results of applying our subscript marking algorithm to XPath expressions.

Example 3

p_1 : `//payment1[creditcard11[cardtype111]]`
 p_2 : `//buyer1/person2[email21[phone22]/name3]
//auction4[annotation41[happiness411]/
author42 = "John"] \square`

Construction of an XPath Condition Tree. An XPath Condition Tree (XCT) is an AND-OR tree representing all logical connectives among conditions specified in an XPath expression. In an XCT, each leaf node corresponds to either a condition or selection path. Two special markers, **sp** and **nr**, are associated

with each leaf node. A path marked with **sp** is a selection path. A path marked with **nr** (non-replaceable) is a path that can not be eliminated or replaced. In particular, all selection and condition paths that are compared against a string value are non-replaceable. For example, in the expression `//a[b = "large"]`, the condition is non-replaceable (but can be unsatisfiable). Given an XPath expression $p = p_1 | \dots | p_k$, an XCT is constructed as follows.

Algorithm 1

Step 1: Create a root node of type OR; create nodes a_1, \dots, a_k , each of type AND, and connect them to the root. If $k = 1$, start with a root node of type AND.

Step 2: For each p_i in p , let ps_i be the selection path and q_1, \dots, q_n be the (possibly nested) predicates. Let $h(q_j)$ be the reduced version of ps_i containing only predicate q_j and the portion of ps_i that precedes q_j . For instance, if $p_i = /a[c]/b[d]/e$, then $h(q_1) = /a[c]$ and $h(q_2) = /a/b/[d]$. Now, for each node a_i , create $n + 1$ children where the first node is marked with **sp/nr** and labeled with the selection path ps_i , and the remaining nodes are labeled with $h(q_0), \dots, h(q_n)$.

Step 3: Expand each node labeled with $h(q_j)$ representing a condition into a subtree according to how the condition is formulated in terms of the logical connectives. For example, if $h(q_i)$ has the pattern `[(a and b) or c]`, expand the node $h(q_j)$ into a subtree rooted in an OR node and having an AND child node.

Step 4: Each leaf node containing a nested predicate of the form `p[qbefore[qinside]/qafter]` is split and two nodes `p[qbefore/qafter]` and `p[qbefore/qinside]` rooted at an AND node are created.

Step 5: Predicates containing string comparison are non-replaceable. Associate a marker **nr** with each leaf node whose label contains such a comparison. \square

An example of an XCT for an XPath expression is shown in Figure 3. An XCT will be modified and simplified during the optimization phase. For this, it is important to keep in mind that the **nr/sp** markers and subscript markings are propagated to the modified/split nodes unless specified otherwise.

5.3 Path Expansion and Optimization

The expansion of wildcards is the first step of the optimization phase. The expansion modifies the XCT such that all wildcards, e.g., the `*` operator and descendant[-or-self] axes (`//`), in the paths described by leaf nodes are expanded. This relatively simple step, which has is well-known in the context of evaluating regular path queries and thus will not be discussed in detail here, can significantly reduce the number of nodes required to visit in a naive path evaluation. For instance, in general, without any index structure, the path `//buyer` requires to visit all nodes in an XML document tree. Expanding the expression to `/auctions/auction/buyer` allows to skip searching any element node in the XML document tree beyond

the depth 3. In our current prototype, we use the Xtrie for wildcard expansion but plan to use a static typing module instead, which should further speed up the expansion process.

Fig. 3(b) shows the XCT where wildcards have been expanded. Then, in the next step, rewrite rules are applied to obtain an optimized combination of non-redundant and minimal (in terms of path length) conditions. For this, the XCT first is translated into disjunctive normal form (DNF). Each path p in a conjunct corresponds to a fully expanded path, marked with **nr** and/or **sp**. Also, with each such path, a PEC $[p]$ is associated. This information can easily be obtained in the context of path expansion, where resulting states are mapped to PECs. In the following, let $exp \rightsquigarrow exp'$ denote the replacement of an expression exp by the expression exp' .

Rule 1 (Path Shortening) If path p is not marked **nr**, it is replaced by the class representative $rep([p])$, i.e., $p \rightsquigarrow rep([p])$. This rule is applied first to each leaf node.

Rule 2 (Path Redundancy) If two paths p_1 and p_2 belong to the same PEC and p_2 is not marked with **nr**, then $p_2 \rightsquigarrow true$. In general, the following simple rules are applied to pairs of paths, depending on how p_1 and p_2 are logically connected.

- a) p_1 and $p_2 \rightsquigarrow p_1$
- b) p_1 or $p_2 \rightsquigarrow p_1$
- c) $not(p_1)$ and $not(p_2) \rightsquigarrow not(p_1)$
- d) $not(p_1)$ or $not(p_2) \rightsquigarrow not(p_1)$
- e) $not(p_1)$ and $p_2 \rightsquigarrow false$
- f) p_1 and $not(p_2) \rightsquigarrow false$
- g) $not(p_1)$ or $p_2 \rightsquigarrow true$
- h) p_1 or $not(p_2) \rightsquigarrow true$

The next two rules utilize the information about PECs encoded in the CIC-Graph.

Rule 3 (Path Implication) If $p_1 \in C_1$, $p_2 \in C_2$, and C_1 implies C_2 , then

- a) p_1 and $p_2 \rightsquigarrow p_1$, if p_2 is not marked **nr**
- b) p_1 or $p_2 \rightsquigarrow p_2$, if p_1 is not marked **nr**
- c) p_1 and $not(p_2) \rightsquigarrow false$.

Rule 4 (Path Contradiction) If $p_1 \in C_1$, $p_2 \in C_2$, and C_1 contradicts C_2 , then p_1 and $p_2 \rightsquigarrow false$.

Rules 1 and 2 obviously utilize the notion of path dependency. Rule 1 states that a condition path can always be replaced by its class representative if it is not marked **nr**. Rule 2 refers to the case where two paths are path-dependent, and one of it is replaceable. Then a redundant condition is found and can be eliminated. Rule 3 states that if the class of a path implies the class of another path, then one of the two paths is eliminated depending on how the conditions are logically connected and whether any of them is replaceable. Finally, if two paths belong to two classes that are contradictory, rule 4 states that a conjunct of the two paths (conditions) is always unsatisfiable.

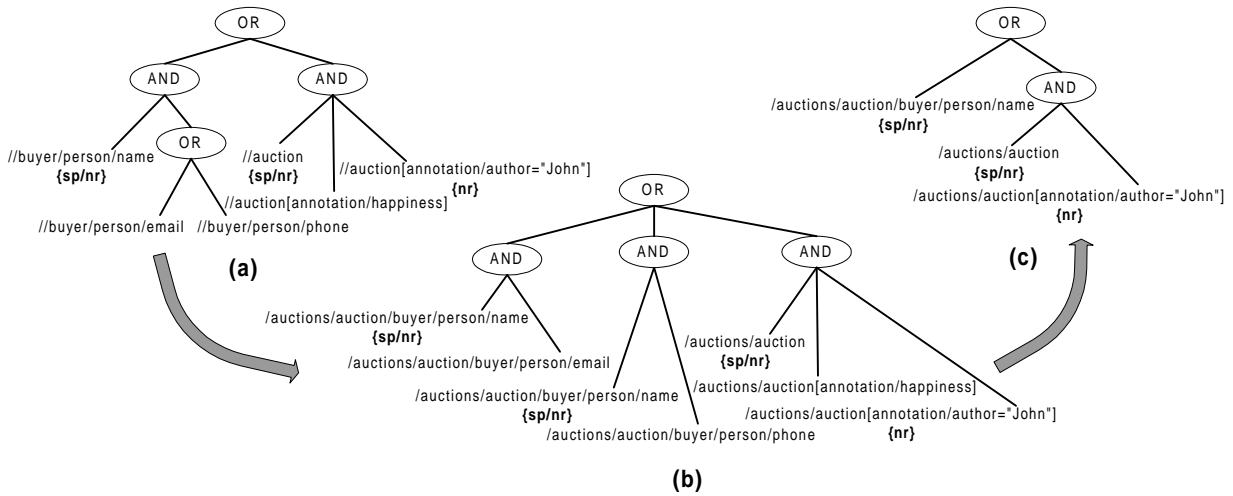


Figure 3: Optimization steps are shown for the expression `//buyer/person[email|phone]/name|//auction[annotation[happiness]]/author = "John"`. Subscripts are omitted for simplicity. The decomposition of the expression is shown in (a), and (b) is the disjunctive normal form of the XCT after wildcard expansion. By applying rewriting rules, we get (c), the optimized expression

5.4 Reconstruction

If the DNF obtained by applying the above rules evaluates to false, the expression is proven unsatisfiable. Otherwise, one can reconstruct an XPath expression from the remaining leaf nodes in the XCT. Since the modified XCT is still in DNF, the paths in each conjunctive term are simply merged into one expression, and the final optimized XPath expression is a union of all such conjuncts.

There is exactly one selection path (marked with **sp**) in each conjunctive term because a selection path is always a necessary component in an XPath expression. If the selection path contains wildcards and is split into multiple selection paths during wildcard expansion, these paths are connected with an OR node, and eventually end up in different conjunctive terms once the tree is translated into DNF. Inserting a condition path into a selection path is as easy as matching the subscripts of the predicates against those in the selection path. The only tricky part is that the paths to be inserted must be in the right order, starting from predicates with the shortest subscript since it is impossible to add a predicate that is supposed to be nested before inserting the outside predicate. For example, by merging `/a1/b2/c3` (selection path), `/a1/b2[d21/f22]`, and `/a1/b2[d21/e211]`, we get `/a1/b2[d21[e211]/f22]/c3`.

Theorem 3 *Given an XPath expression p . The path expression p' obtained through applying (1) path decomposition, (2) wildcard expansion and optimization, and (3) reconstruction is optimized in the sense of Definition 7. ■*

The proof is omitted because of space limitations but can be found in [15]. It should be mentioned, however, that the most important complexity factor

for the algorithm is the pattern of the DNF obtained after wildcard expansion. In our current prototype, we first compare pairs of paths in each conjunct and then for the disjunct. It should also be noted that the optimized expression obtained is not necessarily unique. This is already due to the fact that a PEC can have several shortest paths and thus a representative can be chosen non-deterministically.

6 Evaluation and Experiments

In the following, we first discuss an analytical evaluation scheme that establishes important relationship between different properties of DTDs and PECs that can be derived from the DTDs. The evaluation indicates important properties a DTD should exhibit in order for the our optimization approach to be effective. Then, we outline a performance study we conducted in the context of different systems where a mix of XPath expressions was evaluated against XML documents of varying sizes.

Properties of DTDs and associated PECs. The ratio between required and optional elements in a DTD clearly has an impact on how many PECs can be derived from a DTD. If all elements in a DTD are required, there is obviously only one PEC which contains all the possible paths. For path expressions containing conditions, most probably the conditions are redundant and can be removed. On the other hand, if most elements are optional, there are only few path dependencies and implications, and the total number of PECs is relatively large. In this case, not only the size of the CIC-graph is large compared to the size of the Xtrie, but there is also no gain in grouping paths into PECs because of small class sizes. Most importantly,

the probability of having redundant or simpler conditions is small. In summary, the ratio between required and optional elements in a DTD can be considered as a preliminary indication of the usefulness of our optimization framework. The higher this ratio between required and optional elements, the more effective is our approach.

The above ratio can also be considered as an approximation of the number of PECs for a DTD. To better evaluate the usefulness of our approach, we looked at ratio between possible paths and PECs, which gives the average number of paths per PEC. Intuitively, the larger this ratio, (a) the more rules are implied by a DTD and (b) the smaller the CIC-graph as compared to the size of an XTrie, and thus the more effective is our approach.

Another important factor is the size of an XTrie. In our complexity analysis, we stated that the theoretical upper bound is exponential in the number of elements in a DTD (Section 4). This is due to possible multiple occurrences of the same element in different subtrees. If every element occurs in no more than one content model, the number of nodes in the XTrie is the same as the number elements in the DTD. A large XTrie not only increases storage requirements, but it also implies a larger overhead for the optimization algorithm. Therefore, our approach performs better if an XTrie is of reasonable size (e.g., can easily be kept in main memory). The ratio between the size of an XTrie and the size of DTD thus can then be considered as a second factor in determining the usefulness of our optimization approach. The lower this ratio, the more performance gain in evaluating XPath expressions is implied.

To justify the above claims, we randomly selected 100 DTDs from the OASIS repository at www.oasis-open.org for analysis. They are from different application domains, such as finance, medical/health-care, and many e-commerce related areas. Out of these, 37 were recursive. The number of elements in these DTDs ranges from 8 to 1423, and the depth from 3 to 17. The numbers of elements, admissible paths, and PECs for 62 non-recursive DTDs are shown in Figure 4. One DTD with over 3 million possible paths was omitted. Obviously, the number of PECs is always less than the number of possible paths. It is interesting to note that the number of PECs is less than the number of elements in a few DTDs, which demonstrates the high degree of compactness of a CIC-graph as an abstract representation of path dependencies.

The number of path implications, class implications and i-edges in the CIC-graph for the DTDs are compared in Figure 5. For most DTDs, the number of class implications is significant lower than the number of path implications. Such difference can be considered as the gain in grouping paths into PECs. Since

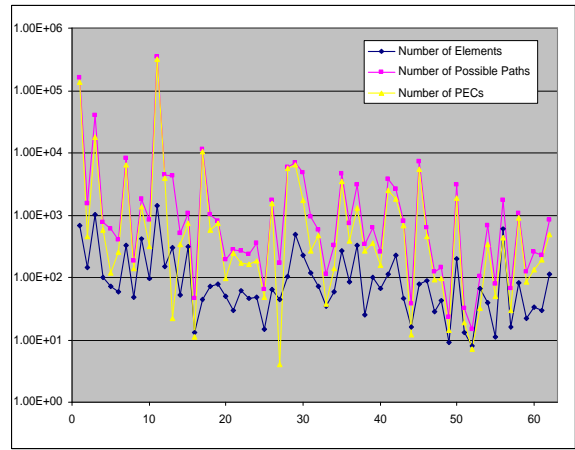


Figure 4: Number of elements, admissible paths, and PECs for DTDs

our approach makes use of the transitivity property of path/class implication in the construction of the CIC-graph, the number of i-edges is even lower than the number of class implications. A similar comparison was done for contradiction, and shows a similar result. Figure 6 gives the two usefulness factors for the sample

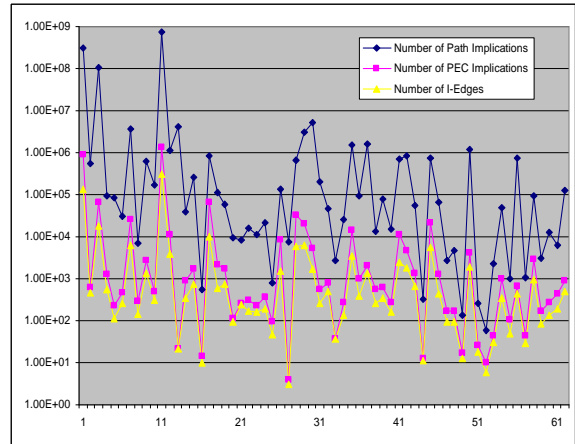


Figure 5: Number of path implications, PEC implications and i-edges

DTDs. The average number of paths per class is 5.5, the average growth from a DTD to an XTrie is 23.45. **Evaluation of XPath Expressions.** We implemented the algorithms presented in this paper and conducted experiments in the context of different systems that provide XPath to query a collection of XML documents. In our experiments, we investigated XML documents of varying size. The documents have been generated using the tool provided as part of the XML Benchmark Project [19], and ranged from 2MB to 16MB in size. Out of a large mix of queries, two queries and their optimized version shown below were used as characteristic queries for performance

evaluation.

Q1. `/site/people/person[email]/name`
 \rightsquigarrow `/site/people/person/name`
 Q2. `//auction[buyer/date and buyer/time]`
 \rightsquigarrow `/site/auctions/auction[buyer]`

The queries were evaluated using two different XML database systems, XIS from eXcelon Corp. (www.exceloncopr.com) and eXist (exist.sourceforge.net). XIS stores XML documents in eXcelon’s ObjectStore object-oriented database. Documents are mapped to proprietary, B-tree-like structures. eXist is an Open Source native XML database that stores documents in an internal, native XML-DB. The sample queries were also evaluated using Jaxen (jaxen.org), a Java XPath engine. The results showing the improvement ($time_{new}/time_{old}$) in evaluating the queries are shown in Figures 7 and 8.

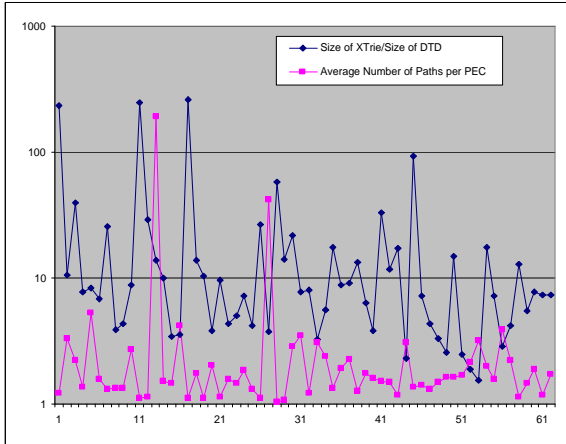


Figure 6: Average number of paths per PEC and ratio between size of Xtrie and size of DTD

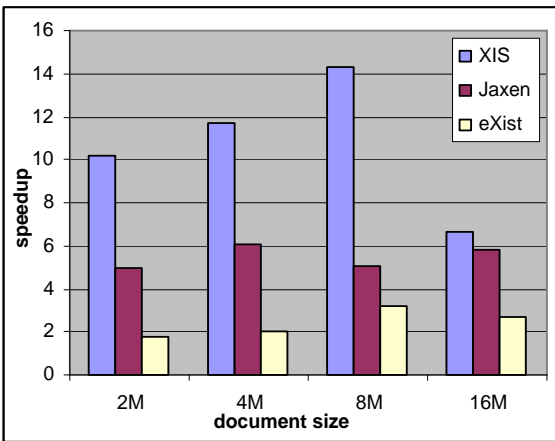


Figure 7: Query 1 (Q1)

Note that Q1 contains only one redundant condition and already gives a drastic performance gain (see Figure 7). Our results show a significant performance gain for both queries in all three systems. The gain is even higher for queries that contain multiple (nested) and possibly redundant conditions. For Q2, Jaxen, which uses the naive traversal approach for evaluation, shows a significant improvement (a factor of 125.2-201.3). This is due to the wildcard “//” in the expression, as it always implies the traversal of the whole DOM tree for the naive approach. Since the other two systems index XML documents in some special way, the performance does not improve as much compared to Jaxen.

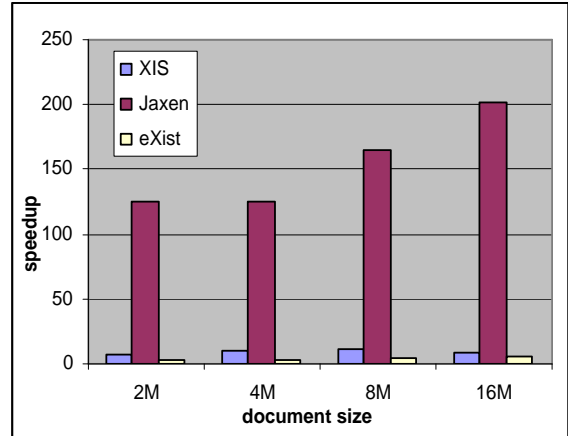


Figure 8: Query 2 (Q2)

7 Conclusions and Future Work

In this paper, we presented a comprehensive framework for the logical, schema-based optimization for a wide range of XPath expressions. In particular, it is the first work that considers the optimization of XPath expressions in the context of DTDs and it thus complements existing work on query containment. The core idea of our approach is to determine and utilize relationships among paths that are possible in documents conformant to a given DTD. Such relationships, represented by path equivalence classes, are aggregated in a concise fashion in a CIC-graph. Such a graph is used at query compile time to determine (1) redundant conditions, (2) unsatisfiable combinations of conditions (and selection paths), and (3) simplification of conditions in XPath expressions. We also detailed important characteristics a DTD should exhibit in order for the optimization framework to be effective. We discussed several types of ratios that can be employed at schema compile time to determine whether employing our optimization scheme will likely result in a performance gain for evaluating XPath expression. It is important to note that, although not detailed in this paper, recursive DTDs can be handled in a similar way

and cause only a minor overhead at query compile time [15].

Although our framework covers a wide range of XPath expressions, because of space limitations, we have not considered optimizations for the full fledged XPath specification. For example, attribute and text node tests have not been addressed explicitly. It should, however, be obvious that condition and selection paths preceding such tests can be handled in the same way as presented in our approach. We are currently extending our framework to be able to deal with a much wider range of XPath expressions. In particular, we are investigating how document order and thus the remaining axes in XPath can be considered in our framework.

An important task of our current research is to further optimize the matching of (expanded) paths to path equivalence classes and to investigate compression techniques for CIC-graphs and XTries, respectively. Preliminary experiments using different hashing techniques (e.g., for path lookups) show promising results. Naturally, logical query optimization, although it can be used independently of a physical query optimization engine, can utilize some information about storage and access structures for XML documents. In particular, we are investigating the integration of information about index structures that might exist for an XML document collection. An interesting open question also is whether XTries and path equivalence classes can provide useful input on creating index structures for XML documents.

References

- [1] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, D. Srivastava: Minimization of Tree Pattern Queries. In *SIGMOD Int. Conf. on Management of Data*, 497–508, ACM, 2001.
- [2] K. Böhm, K. Aberer, M. T. Özsu, K. Gayer: Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition. In *IEEE Forum on Research and Technology Advances in Digital Libraries*, 196–205, IEEE Computer Society Press, 1998.
- [3] P. Buneman, W. Fan, S. Weinstein: Interaction between Path and Type Constraints. In *PODS 1999*, 56–67, ACM, 1999.
- [4] P. Buneman, W. Fan, S. Weinstein: Path Constraints in Semistructured Data. *J. of Computer and System Sciences* 61:2 (Oct.2000), 146–193.
- [5] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu: XQuery 1.0: An XML Query Language. W3C Working Draft, W3C, Dec 2001. <http://www.w3.org/TR/xquery>.
- [6] J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0, W3C Recommendation, November 1999. www.w3.org/TR/xpath.
- [7] D. Calvanese, G. De Giacomo, M. Lenzerini: Representing and Reasoning on XML Documents: A Description Logic Approach. *Journal of Logic and Computation* 9:3 (1999), 295–318.
- [8] J. Clark: XSL Transformation (XSLT), version 1.0. W3C Recommendation, W3C, 1999.
- [9] A. Deutsch, V. Tannen: Containment of XPath Expressions Under Integrity Constraints. Tech. Report MS-CIS-01-21, Univ. of Pennsylvania, 2001.
- [10] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, P. Wadler: XQuery 1.0 Formal Semantics. W3C Working Draft, W3C, June 2001. www.w3.org/TR/xquery-semantics.
- [11] D. Florescu, A. Levy, D. Suci: Query Containment for Conjunctive Queries With Regular Expressions. In *PODS 1998*, 139–148, ACM, 1998.
- [12] M. Fernandez, J. Marsh: XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, W3C, Dec 2001. www.w3.org/TR/query-datamodel/.
- [13] M. Fernandez, D. Suci: Optimizing Regular Path Expressions Using Graphs Schemas. In *Fourteenth Int. Conf. on Data Engineering*, 14–23, IEEE Computer Society Press, 1998.
- [14] R. Goldman, J. Widom: Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, 436–445, 1997.
- [15] A. Kwong, M. Gertz: Schema-based Optimization of XPath Expressions. Techn. Report, University of California, 2002.
- [16] T. Milo, D. Suci: Index Structures for Path Expressions. In *7th International Conference on Database Theory*, 277–295, LNCS, Springer, 1999.
- [17] J. McHugh, J. Widom: Optimizing Branching Path Expressions. Tech. Report, Dep. of Computer Science, Stanford University, 1999.
- [18] J. McHugh, J. Widom: Query Optimisation for XML. In *Int. Conf. on Very Large Data Bases*, 1999.
- [19] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, R. Busse: The XML Benchmark Project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatics, The Netherlands, April 2001.
- [20] P. Wadler. Two Semantics for XPath, 2000, www.research.avayalabs.com/users/wadler/topics/xml.html