On Testing Satisfiability of Tree Pattern Queries

Laks V.S. Lakshmanan, Ganesh Ramesh,
Hui (wendy) Wang, Zheng (Jessica) Zhao
Department of Computer Science
University of British Columbia
{laks,ramesh,hwang,zzhao}@cs.ubc.ca

Abstract

XPath and XQuery (which includes XPath as a sublanguage) are the major query languages for XML. An important issue arising in efficient evaluation of queries expressed in these languages is *satisfiability*, i.e., whether there exists a database, consistent with the schema if one is available, on which the query has a non-empty answer. Our experience shows satisfiability check can effect substantial savings in query evaluation.

We systematically study satisfiability of tree pattern queries (which capture a useful fragment of XPath) together with additional constraints, with or without a schema. We identify cases in which this problem can be solved in polynomial time and develop novel efficient algorithms for this purpose. We also show that in several cases, the problem is NP-complete. We ran a comprehensive set of experiments to verify the utility of satisfiability check as a preprocessing step in query processing. Our results show that this check takes a negligible fraction of the time needed for processing the query while often yielding substantial savings.

1 Introduction

With XML becoming the standard for data exchange, substantial work has been done on XML storage, and query processing and optimization [17, 6, 19, 7, 15, 1, 18, 12, 23, 13]. However, relatively little work has been done on detecting whether a given query is *satisfiable*, i.e., whether there is any database satisfying the query. This is an important problem for the following reasons. (1) Formulating queries against XML databases can be more challenging than for relational databases. As a preview, we will show by example, how *very similar queries can greatly vary in terms of satisfiability*. Indeed, even in the context of relational databases, Levy et al. [8] studied satisfiability for various fragments of the datalog query language and established complexity and decidability results. Hidders [5] is the only work on XPath satisfiability we are aware of. A detailed comparison with our work appears in Section 6. (2) XML is intended to cater for situations where no a priori schema is available for data. Querying an XML database in the absence of any schema knowledge can be tricky. The interaction between various structural constraints, that restrict structural relationships among elements, and value-based constraints, that constrain the contents of elements or their attribute values, can be intricate. (3) Even when a schema is known, getting the query right can still be non-trivial for the user. For, the schema imposes structural constraints on its own which tend to interact with structural and value-based constraints in the query in subtle ways and may make the query unsatisfiable. Our experience shows that checking satisfiability of queries can pay substantial dividends in saving considerable time in query evaluation, while adding a negligible overhead to the overall query evaluation. Besides, given the considerable similarity between a satisfiable query and an unsatisfiable one, it would be useful to have the system assist the user in getting their queries right. Satisfiability testing is a necessary first step in building any such tool. This was the motivation behind our work. Next, we shall illustrate these points with examples.

XQuery [3] is the de facto standard query language for XML and includes XPath [3] as a sublanguage. Both these languages are based on a basic paradigm of finding bindings of variables by matching tree patterns against a database. Benedikt et al. [9] study the expressive power of tree pattern queries in relation to XPath and existential first-order logic. E.g., consider the XPath expression `//a[/b//d = /c//d]`. It corresponds to the tree pattern query (TPQ) Q_4 in Figure 1 (ignore dashed lines for now). Single (double) lines represent parent-child (ancestor-descendant) relationship between nodes.[1] As another example, consider the XQuery statement:

```
FOR   $a IN document(‘‘doc.xml’’)//a,
```

[1]TPQs are formally defined in Section 2.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

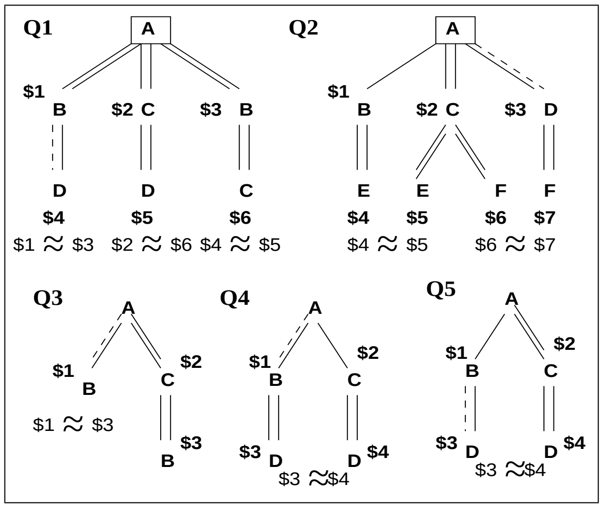


Figure 1: Examples in the Absence of Schema

```
      $e IN $a/b//e, $f IN $a/d//f,
      $c IN $a//c, $e1 IN $c//e, $f1 IN $c//f
WHERE $e = $e1 AND $f = $f1
RETURN{$a}
```

This corresponds to the TPQ Q_2 in Figure 1. Indeed, each TPQ in the figure corresponds to an XPath expression, or an XQuery query. Each query is unsatisfiable, but when the dashed lines are added (i.e., when certain parent-child relationships are relaxed to ancestor-descendant), the queries become satisfiable. We explain this below.

First, consider Q_3. It asks for `A` nodes that have a node `B` that is both a child and is a descendant via an intermediate node `C`. This is clearly unsatisfiable in any tree. However, relaxing the child to descendant makes it satisfiable. Next, consider Q_4, which asks for `A` nodes which have a descendant `D` via a `B` child as well as via a `C` child. Since each element has a unique tag, this is unsatisfiable. Again, relaxing one of the child constraints to descendant renders the query satisfiable. Q_5 is unsatisfiable, but for a subtler reason. If the two `D` nodes are the same, the `C` descendant of `A` must be a descendant of the `B` child. The descendant `D` of the `C` node must thus be at a distance 3 or more whereas the `D` child of `B` is at a distance 2 from `A`, which is impossible. Once again, relaxing any of the child constraints to descendant renders the query satisfiable.

Next, consider Q_2. The constraint that the two `E` leaves must be identical requires nodes `A, B, C` and the two `E` nodes to lie on the same (root-to-leaf) path. Similarly, the identity of the two `F` nodes requires nodes `A, B, C` and the two `F` nodes to lie on the same path. This is impossible, since `C`, having a different tag than the two children of `A`, is forced to be a descendant of both, whereas `B` and `D` cannot lie on the same path. Relaxing the child constraint on `A, D`, e.g., to descendant makes the query satisfiable. Finally, consider Q_1. All edges except on `B, D` are descendant constraints. The query is unsatisfiable because the two `B` nodes are the same node, say ν, and ν has a child `D`, which is a descendant of a descendant `C` node of ν. A contradiction arises because of the inconsistency in the required distance between ν and the `D` node. Again, relaxing the only child constraint in the query to descendant renders it satisfiable. A general remark about all queries is instead of relaxing a child constraint, dropping any other constraint in the query (e.g., identity of two nodes) also renders it satisfiable.

The examples show that reasoning about satisfiability is interesting and non-trivial. We make the following contributions in this paper.

- Reasoning about satisfiability can be reduced to making inferences about relationships between nodes and/or their contents or attribute values. We develop inference rules for deducing additional structural relationships between query nodes from those stated in the query (Sections 3 and 4).
- We propose a *constraint graph* for a tree pattern query. It consists of a structural part that captures structural constraints in the query and a value-based part that captures value-based constraints. Using our inference rules, we develop a *chase procedure* for closing the constraint graph w.r.t. all constraints implied by given ones. We show the chase is complete when the query contains no wildcards: a query is satisfiable iff its constraint graph, when chased, does not result in any violations, in a precisely defined sense. Our inference rules and chase are developed for both when no schema is known and when a schema is given (Sections 3 and 4).
- We identify conditions under which testing satisfiability is NP-complete (Sections 3.3 and 4.2.1) and when it can be done in polynomial time. For the latter cases, we develop efficient algorithms for satisfiability testing (Sections 3.2 and 4.2).
- Finally, we ran a comprehensive set of experiments on a synthetically generated data set on several well-known DTDs including `auction.dtd` and `protein.dtd`, tested various kinds of satisfiable and unsatisfiable queries, and measured both the additional overhead incurred on satisfiable queries and the amount of savings on unsatisfiable queries. While the savings are more than an order of magnitude, our results show that the overhead is a small fraction of overall query evaluation time (Section 5).

Some basic definitions and a problem statement are given in Section 2. Related work appears in Section 6, while Section 7 summarizes the paper and discusses future work.

2 Background and Problems Studied

A(n XML) *database* is a finite rooted ordered tree $D = (\mathcal{N}, \mathcal{E}, r, \lambda)$, where $\mathcal{N}$ represents element nodes, $\mathcal{E}$ represents parent-child relationship, λ, the labeling function, assigns a tag with each node, and r is the root. Associated with each node is a set of attribute-value pairs. In this paper, we do not consider order any further. Fig. 2(a) shows an example database D.

Tree pattern queries, introduced in [1], capture a useful fragment of XPath. A *tree pattern query* (TPQ) is a triple $Q = (V, E, F)$, where (V, E) is a rooted tree, with nodes V labeled by variables, and with $E = E_c \cup E_d$ consisting of two kinds of edges, called pc- (E_c) and ad-edges (E_d), corresponding to the child and descendant axes of XPath. A distinguished node in V (shown boxed in Figure 1)[2] corresponds to the answer element. F is a conjunction of tag constraints (TCs), value-based constraints (VBCs), and node identity constraints (NICs). TCs are of the form $\$x.tag = t$, where t is a tag name. VBCs include selection constraints $\$x.val$ relop c, $\$x.attr$ relop c, and join constraints $\$x.attr$ relop $\$y.attr'$, and $\$x.val$ relop $\$y.val$, where relop $\in \{=, \neq, >, \leq, \geq, <\}$, *attr, attr'* are attributes, *val* represents content, and c is a constant. *With a few clearly identified exceptions, we assume no disjunctions appear in VBCs, throughout the paper. When we do allow disjunctions, they are confined to selection conditions.* NICs are $\$x$ idop $\$y$, where idop $\in \{\approx, \not\approx\}$.[3] We adopt the term structural constraints to refer to NICs and predicates of the form $pc(\$x, \$y), ad(\$x, \$y)$, representing pc- and ad-edges. Figure 2(b) shows an example TPQ Q. An example use of disjunction in selection constraints is the constraint $\$3.type$ = ‘paperback’$\vee$ $\$3.type$ = ‘spiralbound’ on node \$3 in place of the existing constraint there.

Answers for TPQs are formalized using matchings. A *matching* of a TPQ Q to a database D is a function $h : Q \to D$ that maps nodes of Q to nodes of D such that: (i) structural relationships are preserved – whenever $(x, y) \in E_c$ $h(y)$ is a child of $h(x)$ in D and whenever $(x, y) \in E_d$, there is a path from $h(x)$ to $h(y)$ in D; and (ii) the formula F is satisfied. We say that a database D *satisfies* a query Q provided there is a matching $h : Q \to D$. A matching of the query Q in Figure 2(b) to the database D of Figure 2(a) is schematically illustrated with numbers besides database nodes. A query Q is *satisfiable* provided there is a database D that satisfies Q.

For readability, whenever the tag constraint $\$x.tag = t$ appears in a TPQ $Q = (V, E, F)$, we drop that constraint from the formula part F, and write t right next to node $\$x$ in Q. If node $\$x$ is not tagged, we associate a wildcard ‘*’ next to node $\$x$ in Q. This is illustrated for query Q in Figure 2(b)-(c). This corresponds to the XPath expression `/bib[//*/text()=‘Raymond Smullyan’]/*[@type=paperback]/author[text()=‘B. Russel’]`. If the constraint $\$3.type \neq$ paperback were replaced by $\$3.type =$ paperback in Q, database D in Figure 2(a) wouldn’t satisfy it, as no matching

[2]Since distinguished nodes do not play any role in satisfiability, we do not consider them further.

[3]The constraints $\$x \approx \$y$ and $\$x \not\approx \$y$ say nodes $\$x$ and $\$y$ are (not) the same.

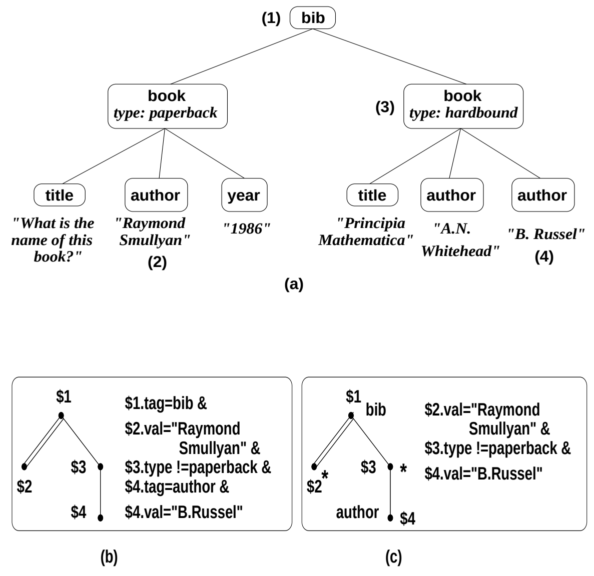


Figure 2: An example: (a) database D and: (b) TPQ Q, (c) Q made more readable.

is possible. In the sequel, we write element tags or wildcards next to query nodes as appropriate. Thus, a query Q is said to have wildcards if one or more nodes do not have their tags constrained by a TC. Otherwise it *wildcard-free*. Q is *join-free* if it contains no join constraints and no NICs.

We abstract the schema of a database (in our paper, we only consider DTDs) as a graph with nodes corresponding to tags and edges labeled by one of the quantifiers ‘?, 1, *, +’ with their standard meaning of ‘optional’, ‘one’, ‘zero or more’, and ‘one or more’ respectively. An example of a schema graph appears in Figure 8. It says, e.g., that `categories` consists of `category` elements, each of which has a unique `description`.

Problems Studied: We consider testing satisfiability of various classes of TPQs (with/without VBCs, with/without disjunction in VBCs, with/without join and node identity constraints, with/without wildcards) both in the absence of a schema and in the presence of a schema without disjunction (i.e., choice) and cycles.

3 Satisfiability without Schema

Given a TPQ Q, determining whether Q is satisfiable in the absence of a schema, solely depends on the structural constraints and any VBCs present in Q. In addition, it may be necessary to consider disjunctions and wildcards in the query, if present. We systematically study the problem for various TPQ classes.

3.1 Join-free TPQs with Wildcards

Recall that join-free TPQs do not contain join or node identity constraints. Note that Q may still involve value-based selection constraints. In the special case that Q has no VBCs, it is always satisfiable. Indeed, a satisfying instance D for Q can be constructed as follows. D is a tree isomorphic to the query tree Q except all edges are pc-edges. For every query node that is tagged, the corresponding node in D has the same tag; if the query node is a wildcard, the corresponding node in D may have an arbitrary tag. It is easy to see that D always satisfies Q.

Suppose $\mathsf{Q} = (\mathsf{V}, \mathsf{E}, \mathsf{F})$ does contain VBCs. Since it does not contain any join constraints, every VBC constrains a unique node in Q. Let F_x be the maximal subformula of F that constrains node x. To verify that Q is satisfiable, it then suffices to verify if F_x is satisfiable for each node x. The following proposition summarizes the situation for join-free TPQs.

Proposition 3.1 *For a join-free tree pattern query* Q*, possibly containing wildcards, the following holds:*

1. *If* Q *contains no VBCs associated with any node, then* Q *is satisfiable.*
2. *If* Q *contains value-based selection constraints (but is join-free), then* Q *is satisfiable iff for every node, the associated set of VBCs is consistent.* ▮

The complexity of verifying satisfiability thus depends on the kinds of formulas F_x constraining each node x. If no disjunction occurs, consistency of F_x and hence of F can be verified in polynomial time using the sound and complete axiom system given in [21]. If VBCs F_x associated with a node x can involve arbitrary disjunctions, testing consistency of F_x becomes equivalent to SAT and hence is NP-complete. If F_x is a disjunction of conjunctions, then the method proposed in [21] can be easily extended to yield a polynomial time test for satisfiability of Q.

3.2 Wildcard-free TPQs with Joins

Let Q be a TPQ containing join and/or node identity constraints, but no wildcards and no disjunction. We relax the latter restrictions in Section 3.3. The presence of join and node identity constraints interacts in an intricate way with the structural constraints. E.g., the constraint $x{\approx}y$ implies any ancestors of x and y in the query Q must lie on the same path in a satisfying database. Below, we separate the reasoning into structure and value-based parts and pin down exactly how they handshake.

3.2.1 Reasoning about Structure

In this section, we consider queries with just NICs. The effect of VBCs of the form $\$x.val$ relop $\$y.val$ etc. are addressed in Section 3.2.2. Some issues involved in reasoning about satisfiability are illustrated by the following example.

Example 3.1 [Structural reasoning]
Consider the query in Figure 3, which is identical to query Q2 in Figure 1. As discussed in the introduction, it is unsatisfiable. The reasoning involves inferring that node pairs \$2 and \$4 must lie on the same root-to-leaf path as well as that they must be cousins of each other, leading to a contradiction. ▮

The example illustrates several points. **1.** Testing satisfiability involves inferring relationships between pairs of nodes based on structural constraints stated in the query. Thus, we need inference rules. **2.** Some of the intermediate relationships inferred above cannot be directly represented in the language of TPQs (e.g., "x and y must lie on the same path"). Thus, the language is not "closed" w.r.t. satisfiability reasoning. We could represent the new relationships by permitting disjunction in structure. E.g., x and y lie on the same path iff $(x \approx y \vee \mathsf{ad}(x,y) \vee \mathsf{ad}(y,x))$. However, permitting arbitrary disjunctions can lead to high complexity. We show that all we need to do is add the following predicates: $\mathsf{sad}(x,y)$ meaning $x{\approx}y$ or $\mathsf{ad}(x,y)$, $\mathsf{OTSP}(x,y)$ meaning $\mathsf{sad}(x,y)$ or $\mathsf{ad}(y,x)$, $\mathsf{COUS}(x,y)$ meaning $\neg\mathsf{OTSP}(x,y)$. Note that the predicates $\mathsf{OTSP}, \mathsf{COUS}, \approx, \not\approx$ are symmetric while $\mathsf{pc}, \mathsf{ad}, \mathsf{sad}$ are not. This expanded set of predicates is indeed closed w.r.t. satisfiability reasoning.

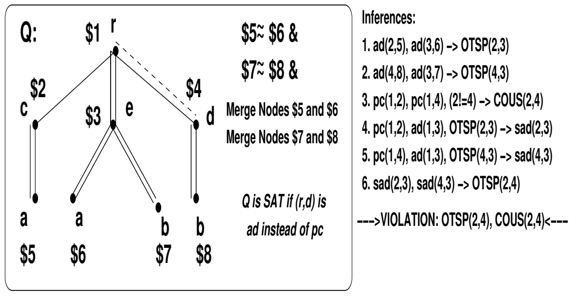

Q:
$1
r
$2
c
$3
e
$4
d
a
a
b
b
$5
$6
$7
$8
$5~ $6 &
$7~ $8 &
Merge Nodes $5 and $6
Merge Nodes $7 and $8
Q is SAT if (r,d) is ad instead of pc
Inferences:
1. ad(2,5), ad(3,6) -> OTSP(2,3)
2. ad(4,8), ad(3,7) -> OTSP(4,3)
3. pc(1,2), pc(1,4), (2!=4) -> COUS(2,4)
4. pc(1,2), ad(1,3), OTSP(2,3) -> sad(2,3)
5. pc(1,4), ad(1,3), OTSP(4,3) -> sad(4,3)
6. sad(2,3), sad(4,3) -> OTSP(2,4)
--->VIOLATION: OTSP(2,4), COUS(2,4)<---

Figure 3: Inferring Structural Predicates

Determining satisfiability of a query works as follows. First, we use inference rules to obtain the closure of structural predicates. Then, we check the resulting set of predicates for violations (defined below). The query is satisfiable iff the set of predicates is violation-free (consistent).

Structural Constraint Graph: In order to efficiently implement a procedure for satisfiability checking, we construct a *(structural) constraint graph* $\mathsf{G_Q}$ for the query Q as follows. $\mathsf{G_Q}$ contains one node for each query node. For each predicate $\phi(x,y)$ in Q, $\mathsf{G_Q}$ contains a directed edge labeled ϕ from x to y. For symmetric predicates, the edge is bidirected.

Inference Rules and Chase: New structural predicates are inferred from existing ones in the query by using a set of *inference rules.* An inference rule is of the form $\mathsf{P}_1, \ldots \mathsf{P}_k \rightarrow \mathsf{R}$ and says "if predicates $\mathsf{P}_1, \ldots, \mathsf{P}_k$ are true, then R is true". Inference rules are used for achieving closure of structural predicates and thus for catching inconsistencies caused by conflicting pairs of predicates. For the structural predicates, we have developed a total of 22 inference rules.[4] For brevity, we show only some interesting rules in Figure 4 and explain some selected ones. The complete details can be found in [11]. We explain three of the rules. Rule **2** says whenever x lies on the same path as each of a pair of cousins y and z, then x must be their ancestor[5]. Rule **3** says two unequal nodes x, y at an equal distance from a node z must be cousins. The equal distance implies the paths from z to x and y must involve only pc-edges. Rule **7** says whenever x and y are on the same path, x is a child of an ancestor of y, then y must be a self or descendant of x, i.e., $\mathsf{sad}(x, y)$. The *chase procedure* is to simply apply the inference rules until no new inferences are possible. If a violation, defined next, is detected at any point, we can exit from chase early. We will discuss a more efficient implementation of chase shortly.

1.	$\mathsf{sad}(x,z), \mathsf{sad}(y,z) \rightarrow \mathsf{OTSP}(x,y)$
2.	$\mathsf{OTSP}(x,y), \mathsf{OTSP}(x,z), \mathsf{COUS}(y,z) \rightarrow \mathsf{ad}(x,y)$
3.	$x \not\approx y \rightarrow \mathsf{COUS}(x,y)$, whenever x, y are at the same distance from their least common query ancestor z.
4.	$\mathsf{pc}(x,z), \mathsf{pc}(y,z) \rightarrow x \approx y$.
5.	$\mathsf{pc}(z,x), \mathsf{pc}(z,y), \mathsf{OTSP}(x,y) \rightarrow x \approx y$.
6.	$\mathsf{ad}(x,z), \mathsf{pc}(y,z) \rightarrow \mathsf{sad}(x,y)$.
7.	$\mathsf{pc}(z,x), \mathsf{ad}(z,y), \mathsf{OTSP}(x,y) \rightarrow \mathsf{sad}(x,y)$.

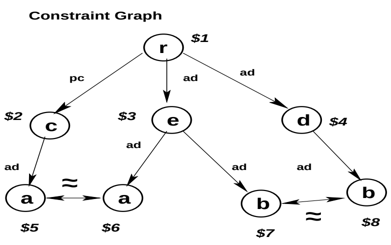
Figure 4: Selected Inference Rules (no schema).

Violations: A *violation* is a pair of conflicting predicates between a pair of nodes. Examples of conflicting pairs of predicates are $x \approx y, x \not\approx y$; $\mathsf{ad}(x,y), \mathsf{sad}(y,x)$; and $\mathsf{OTSP}(x,y), \mathsf{COUS}(x,y)$. Indeed, these three pairs capture all possible violations, since other violations are subsumed by them. For instance, $\mathsf{pc}(x,y)$ conflicts with $\mathsf{COUS}(x,y)$. But since $\mathsf{pc}(x,y)$ implies $\mathsf{OTSP}(x,y)$ this conflict is covered by the pair $\mathsf{OTSP}(x,y), \mathsf{COUS}(x,y)$. Violations make the query unsatisfiable.

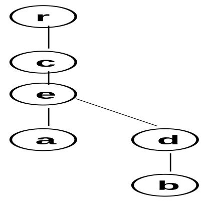
Figure 3 demonstrates the chase as logical inferences. At the end of step 6, we find a violation because of the conflicting predicates $\mathsf{OTSP}()$ and $\mathsf{COUS}()$. To implement the chase more efficiently, we employ the constraint graph. Specifically, given a TPQ Q, we initialize its constraint graph CG_Q. For every pair of nodes $\$i, \$j$, whenever their tags are different, we add a bidirected edge labeled $\not\approx$ between $\$i$ and $\$j$. We apply the inference rules repeatedly. Whenever predicate $p(\$i, \$j)$ is derived, add a (directed) edge from $\$i$ to $\$j$ labeled p if p is one of $\mathsf{ad}, \mathsf{sad}$ and make it bidirected if p is one of $\not\approx, \mathsf{OTSP}, \mathsf{COUS}$. When $\$i \approx \$j$ is derived, we merge nodes $\$j$ and $\$j$. We repeat until no new inferences are made or a violation is detected. A constraint graph, with chase applied on it, is a *chased* constraint graph. Here is an example. The query of Figure 3 becomes satisfiable if the pc-edge from $1 to $4 is changed to an ad-edge (shown dotted in the figure). Figure 5(a) shows the constraint graph for this query and Figure 5(c) shows the chased constraint graph. Figure 5(b) shows a satisfying instance of the query.

[4]Including "trivial" ones such as $\mathsf{pc}(x,y) \rightarrow \mathsf{ad}(x,y)$.

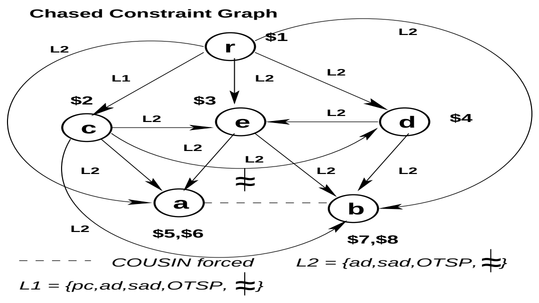
[5]Note that the rule is symmetric



(a) Constraint Graph



(b) Instance



(c) Chased Constraint Graph

Figure 5: Determining Satisfiability

The main result of this section is the following:

Theorem 3.1 (Completeness of Chase) : Let Q be a tree pattern query containing node identity constraints but no wildcards. It is satisfiable iff the chased constraint graph of Q is violation-free. ▌

We refer the reader to [11] for the proof. Here, we give the key intuition. The "If" direction is easy to see since every inference rule is sound and therefore preserves satisfiability. For the "Only If" direction, suppose G is the chased constraint graph of query Q and G is violation-free. We construct a satisfying tree instance as follows.

```
Procedure FastChase(CGraph CG)
   find all $\approx$-classes of nodes;
   for each equivalence class E, find the maximal
      OTSP set as $\bigcup_{x \in E} pred(x)$
      in G, where $pred(x)$ is the set of
      predecessors of x in G;
   for all x, y s.t. $x \not\approx y \in G$, apply the
      distance rule (#3) to derive COUS(x, y);
      propagate COUS() downward using inference rules;
      if COUS(x, y) is derived, add $x \not\approx y$ to G;
      if a violation is found return ``unsatisfiable'';
      if $x \approx y$ is derived, merge x and y;
   while there is no change {
      apply rules for inferring $\approx$, ad, sad;
      if nodes are equated, merge them;
      if a violation is found return ``unsatisfiable''; }
return true;
```

Figure 6: Apply Chase in CGraph

Call a set S of nodes in Q an OTSP set provided $\forall x, y \in S$: $\mathsf{OTSP}(x, y) \in G$. OTSP sets are upward closed, i.e., when $x \in S$, and $\mathsf{sad}(y, x) \in G$, then $y \in S$. Henceforth, we consider maximal OTSP sets, i.e., OTSP sets whose proper supersets are not OTSP sets. The idea is to force relationships between pairs of nodes until G becomes a complete set, i.e., $\forall$ nodes $x, y \in G$ and for any predicate p, either $p(x, y)$ or $\neg p(x, y)$ holds in G. In particular, all nodes in a maximal OTSP sets are totally ordered using a topological sort. Different maximal OTSP sets are incorporated in different branches of the tree.

We next briefly comment on an efficient implementation of the chase. A naive implementation would take time $O(n^5)$, where n is the number of nodes in the query. This is because each rule involves 3 nodes and there are $O(n^2)$ iterations possible in the worst case before no new inferences are made. A more efficient implementation is suggested in Figure 6. The idea is to exploit the upward closure (downward closure) of OTSP (COUS) predicate. It can be shown that maximal OTSP sets can be computed "statically" based on the constraints given in the query. Similarly, we can infer COUS edges efficiently. Inference rules for the remaining predicates need to be applied repeatedly until either a violation is found or no new inferences are possible. The worst-case complexity of this algorithm remains the same. However, in practice it is much better than the naive algorithm.

3.2.2 Interaction with VBCs

Up to this point, we have not considered VBCs. Even when a query is satisfiable w.r.t. its structural constraints, the VBCs may render it unsatisfiable. As mentioned earlier, consistency of a conjunction of VBCs can be checked in polynomial time using the sound and complete axiom system provided in [21]. The checking algorithm can be implemented efficiently using a separate *value-based constraint graph* using ideas similar to the structural constraint graph. The details are similar and are omitted. What about interactions between structural constraints and VBCs? It can be shown that the interaction happens via two main links: (i) The structural constraints may imply $x \approx y$ for nodes x, y. All VBCs applicable to x are applicable to y. This is automatically captured by merging x and y. (ii) VBCs can imply $x \not\approx y$ for nodes x, y. This can in turn trigger inferences of structural predicates.

The procedure for testing satisfiability of a query Q with structural constraints and VBCs is then as follows: (i) Chase the VBCs (using a separate value-based constraint graph); if any violation is found return "unsatisfiable". (ii) Construct the (structural) constraint graph G of Q; propagate all constraints $x \not\approx y$ derived from VBC chase to G and chase it; (iii) Q is satisfiable iff the chase terminates with no violation.

We can show:

Theorem 3.2 (TPQs with VBCs) : Let Q be a tree pattern query with structural constraints and

Disjunction	NICs/join constraints	Wildcards	Complexity
		X	PTIME
	X		PTIME
X	X		NP-Complete
	X	X	NP-complete

Figure 7: Complexity of checking Satisfiability without Schema

VBCs and no wildcards. Then testing satisfiability of Q can be done in polynomial time using the procedure above. ∎

3.3 TPQs with Wildcards, Joins, and Disjunction

We relax the restrictions on TPQs w.r.t. wildcards and disjunctions in this section. The first observation is that when wildcards are allowed, satisfiability testing becomes NP-complete, even when there is no disjunction. This follows from the following result, proved by Hidders [5].

Theorem 3.3 ([5]) : Suppose Q is a tree pattern query with wildcards and only $\approx$ constraints, where the query uses only pc- and sad-edges. Then testing whether Q is satisfiable is NP-complete. ∎

While Hidders' result is couched in terms of a syntactically different language, the fragment for which this result applies corresponds to tree pattern queries with wildcards and $\approx$ constraints, where the entire query reduces to a single maximal OTSP set. It is trivial to adapt his proof for tree pattern queries with regular pc- and ad-edges.

Next, what if we disallow wildcards but allow disjunction in VBCs. The problem again becomes NP-complete.

Theorem 3.4 (TPQs with disjunction) : Let Q be a tree pattern query containing VBCs, with disjunction allowed in selection constraints associated with nodes. Then testing satisfiability of Q is NP-complete. ∎

The proof is by reduction from 3SAT, and only makes use of pc-edges, disjunctive value-based selection constraints, and $\not\approx$ constraints. It continues to hold when $\not\approx$ constraints are replaced by join constraints.

The complexity results for the schemaless case are summarized in Figure 7.

4 Satisfiability in the Presence of Acyclic Schema

A schema provides additional knowledge for inferring structural predicates in a query. E.g, consider the schema and query Q4 in Figure 8. It is not satisfiable. The query Q4 asks for *text* which is both a child of *description* and a descendent of *parlist*, but the schema does not permit this. However, if *text* is changed to be a descendent of *parlist*, Q4 becomes satisfiable. Suppose Q4 is accordingly changed. In the absence of schema, the best we can conclude about *description* and *text* then is that they must lie *on the same path* but using the schema, we can conclude that *description* is the *ancestor of text.* A schema (auction.dtd) together with a set of unsatisfiable queries as well as minor variants which are satisfiable are given in Figure 8. The reader is encouraged to reason about their satisfiability.

In the rest of this section, we consider acyclic (DAG) schema. Extensions to cyclic schemas will appear in the full paper.

4.1 TPQs without VBCs

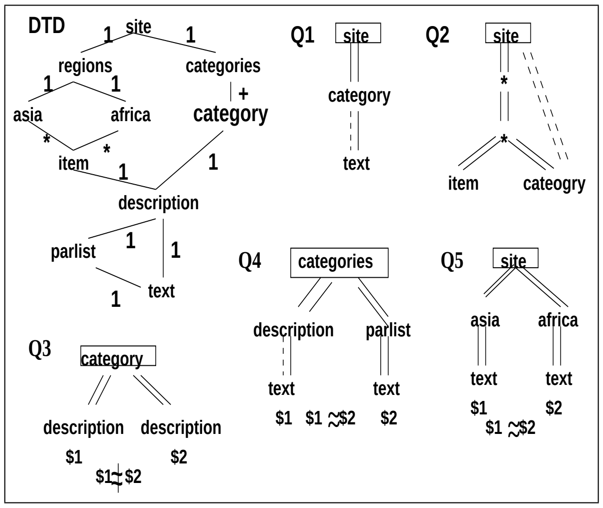


Figure 8: Examples in the Presence of Schema

When no VBCs are present, for a query to be satisfiable with respect to a schema, its structural constraints need to be consistent with the schema. An *embedding* of a query into a schema, defined below, precisely captures this consistency.

Definition 4.1 [Embedding] An *embedding* of a query Q into a schema $\triangle$ is a function $f : Q \to \triangle$ satisfying the following conditions: (i) f maps each tagged node to a node with the same tag; (ii) whenever (x, y) is a pc-edge (ad-edge) in Q, there is an edge (path) from $f(x)$ to $f(y)$ in Δ. ▮

Consider query Q4 in Figure 8, but without the join condition. The reader can verify the existence of an embedding into the schema in Figure 8. In the absence of wildcards, the testing the existence of an embedding reduces to testing for each edge (x, y) in Q with $tag(x) = a, tag(y) = b$ (say), whether an edge or path from a to be exists in Δ, which can be easily tested.

The following result is straightforward:

Proposition 4.1 *Let $\triangle$ be a schema and let Q be a tree pattern query with no wildcards or VBCs. Then Q is satisfiable with respect to $\triangle$ iff there is an embedding f from Q into $\triangle$.* ▮

4.1.1 With Wildcards

Consider the examples in Figure 9. Query Q1 is not satisfiable because no valid instance of the schema can have a path from a to e of length 2. On the other hand, query Q2 is satisfiable because there exists a valid instance of the schema which has a path of length at least 2 from a to e. For each of the queries, the possible schema nodes it could embed to are illustrated as a set, right next to the node in Figure 9. For the node \$3 in query Q1, the set of schema nodes it can embed to is empty. Note that if the query contains *only* wildcard nodes, checking satisfiability trivially reduces to checking if the schema is of a given depth.

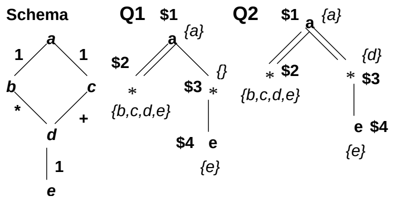


Figure 9: Queries with Wildcards

When wildcards are present, semantically we can assign any tags to the wildcards and check for the existence of an embedding. This approach takes exponential time. What we need is merely confirm the existence of an embedding. This can be accomplished by associating with each query node x a label set $L(x)$. For each tagged node x, we initialize $L(x)$ to be the unique schema node with that tag. For a wildcard node x, we initialize $L(x)$ to be the set of all tags in the schema $\triangle$. We next prune $L(x)$ as follows. First, in a bottom-up phase, we mark all leaves. Whenever all children $y_1, ..., y_k$ of a node x are marked, we delete a tag t from $L(x)$ provided for some y_i, $r(x, y_i)$ holds according to the query Q, where r is pc or ad, but there is no tag $u \in L(y_i)$ such that $\triangle$ contains an edge or path to verify $r(t, u)$. Then mark node x. If at any stage any label set becomes empty, we know the query is unsatisfiable. Once the root is marked, we do a top-down sweep as follows. First unmark the root. For any node x whose parent y is unmarked, delete from $L(x)$ any tag t if there is no tag $u \in L(y)$ such that $r(y, x)$ according to Q, and $\triangle$ contains an edge or path verifying this. Then unmark x. The procedure terminates when an empty label set is detected or when all nodes are unmarked. The pseudocode for this algorithm is shown in Figure 10. We can show:

Theorem 4.1 (Labeling) : Let Q be a TPQ containing wildcards but no VBCs and no NICs. Then Q is satisfiable with respect to a schema $\triangle$ iff for each $x \in Q$, $L(x) \neq \emptyset$, where $L(x)$ is the set of schema labels computed by the procedure in Figure 10. ▮

By precomputing reachability on $\triangle$, given t, t', we can test if $\triangle$ verifies $r(t, t')$, where $r \in \{pc, ad\}$, in constant time. We visit each query node and each query edge at most twice. During each visit, we may

```
CheckLabel(Q,$\triangle$)
For each node x tagged t in Q, $L(x) = \{t\}$
For each wildcard leaf l in Q, $L(l) = \{$tags of $\triangle\}$
Mark all leaf nodes.  Let $r \in \{pc, ad\}$
Repeat { // Bottom-up Phase
$\forall$ nodes $x \in Q$ whose children $y_1, \ldots y_k$ are all marked
   For each child $y_i$ of x {
      Initialize $S_i = \{\}$;
      For each $u \in L(y_i)$ {
         $S_i = S_i \cup \{t' \mid r(t', u) \in \triangle\}$; }
      $L(x) = \bigcap_{i=1}^{k} S_i$; }
      Mark x;
   If L(x) is empty, return(Q is not SAT); }
Until all nodes are marked.
Unmark the root;
Repeat { // Top-down Phase
   For each x whose query parent y is unmarked {
      For each $u \in L(x)$ {
         If $\nexists t' \in L(y)$ s.t.  $r(t', u) \in \triangle$
            remove u from L(x); }
      Unmark x; if L(x) is empty, return(Q is not SAT); }
Until all nodes are unmarked.
return(Q is SAT)
```

Figure 10: Check Wildcard Embedding

need to compare all pairs of tags in the label sets of the two nodes in the edge. Thus, the worst-case time complexity is $O(m^3 + n \times m^2)$, where m is the number of nodes in $\triangle$ and n is the number of nodes in Q.

4.2 Reasoning in the presence of Node Identity Constraints

Let us consider the class of TPQs that contain no wildcards but may contain NICs ($\approx, \not\approx$), and VBCs (without disjunctions). Apart from the interaction between structural predicates and VBCs, there is interaction between schema and the structural constraints imposed by the query.

Example 4.1 [Impact of Schema]
Consider the examples in Figure 11. Query $Q1$ is satisfiable with respect to the schema. The reasoning behind this is as follows. Since nodes \$4 and \$5 are identical, nodes \$2 and \$3 must lie on the same path. From the schema, we can then conclude that \$2 is an ancestor of \$3. Indeed, an instance can be obtained from the chased constraint graph that satisfies the query.

Query $Q2$ is not satisfiable with respect to the schema. Here is why. From the schema, every occurrence of d necessarily has a grandchild f, which is unique. The query asks for two distinct descendents of d tagged f, one as a grandchild and one as any descendent. However, from the schema, we can conclude that nodes \$4 and \$5 are identical i.e., \$4$\approx$\$5 which contradicts the query constraint \$4$\not\approx$\$5.

The example illustrates several points. **1.** The query contains no wildcards. Thus, for two nodes if either sad or $OTSP$ predicate holds, then from the schema, it is possible to conclude a strict *pc* or *ad* relationship between them. Hence, the predicates sad and $OTSP$ which we used in the absence of schema, now become redundant. **2.** We can use the schema to determine when two query nodes are identical. In the schema of Figure 11, there is a *unique path* from node d to f, with all edge labels either '$1'$' or '?''. Hence any two descendents f of a d in the instance should be identical. **3.** Following the same argument, using the schema it is also possible to infer that two nodes must be *cousins*, by determining when the nodes lie on distinct paths.

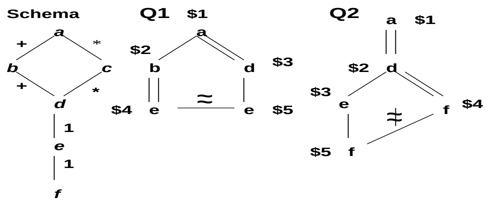


Figure 11: Inference from Schema

Determining satisfiability of a query works as follows. We use the schema to infer structural predicates between any pair of query nodes (which are tagged). We use inference rules to compute the closure of structural predicates and check the resulting set for violations. The query is satisfiable iff the resulting set is violation-free. As before, we use a constraint graph and a set of inference rules to compute the closure, with some differences in the inference rules used.

Inference Rules and Chase: The set of inference rules are adapted from those developed for the schema-less case. Rules involving sad or $OTSP$ are dropped, since the schema allows us to derive an unambiguous ad relationship whenever sad or $OTSP$ holds. Additionally, we need to infer relationships between element types from the schema. The schema can tell us that two tags t, t' are related by a pc-/ad-relationship, or that two query nodes must be identical or that they must be cousins. This static analysis of the schema can be performed using the rules shown in Figure 12, explained next. The complete set of inference rules can be found in [11].

Rule 1 corresponds to "disjoint" nodes. Let x, y be any nodes in a query Q and suppose z is their least common ancestor in Q. Let $\triangle$ be a given schema. Suppose $(z, u_1, ..., u_k, x)$ and $(z, v_1, ..., v_m, y)$ are the paths in Q from z to x and y respectively. We call these paths the query context of x and y. Note that all nodes are tagged in Q. For simplicity, denote the tag of each node by its primed version, i.e., node x has tag x'. Suppose *there is no path in* $\triangle$ *that passes through all the nodes* $z', u'_1, ..., u'_k, v'_1, ..., v'_m, x', y'$ *and in an order compatible with the query contexts above, which respects any pc-relationships present in the query contexts.* Then we can conclude that x *and* y *must be cousins in every valid instance of* $\triangle$, *which satisfies* Q. When this condition holds, we say x' and y' are *disjoint*.

As an example, consider query Q4 in Figure 8, *without* the dashed line added. Then query nodes \$1 and \$2 (with tag `text`) are necessarily cousins. This is because there is no path in the schema that passes through `categories`, `description`, `parlist`, and `text` in any compatible order, such that there is

```
1. whenever x, y are disjoint, infer COUS(x, y).
2. whenever z is lca(x, y), x and y are unique w.r.t.  z,
   the path from z′ to x′ that satisfies the query
   context of x is identical to the path from z′ to y′
   that satisfies the query context of y,
   tag(x) = tag(y), infer ≈(x, y)
3. whenever z is the lca(x, y), x and y are unique w.r.t.
   z, the unique path from z′ to y′ that satisfies the query
   of x and y contains edge (x′, y′), tag(x) ≠ tag(y),
   infer pc(x, y)
4. whenever ad(x, z), ad(y, z), △:  exactly one path from
   x′ to y′ and that path is an edge, infer pc(x, y)
```

Figure 12: Selected Inference Rules(with Schema)

a direct edge from description to text, so $1′ and $2′ are disjoint. However, if the edge is relaxed to an ad-edge (i.e., dashed line is added), such a path exists in the schema, so $1′ and $2′ are not disjoint, hence $1 and $2 are not necessarily cousins. Disjointness can be checked efficiently using a variant of merge sort and in time linear in the sum of sizes of the two query contexts.

Rules 2-3 correspond to "unique" nodes. Let Q be a query and x and y be the nodes in Q, such that $ad(y, x) \in G$. Let △ be a schema. Then x is *unique* with regard to y whenever △ has exactly one path from y' to x' satisfying the query context of x and y, and no edge on this path is labeled * or +. The intuition behind rule 2 is that the query paths from z to x as well as from z to y will both be mapped necessarily to one path in every valid instance of △. So, if x and y have the same tag, they must map to the same instance node. Rule 3 has a similar intuition.

Rule 4 says whenever $ad(x, z), ad(y, z)$ holds, clearly one of x, y must be a parent/ancestor of the other (when x and y have different tags). This is determined by the schema.

Finally, the chase procedure for TPQs (with NICs but no wildcards) in the presence of a schema is as follows.

- First, construct the constraint graph G of Q as for the schemaless case.
- Next, using static analysis of the schema, infer all COUS, ≈, pc, ad relationships and add them to G.
- Chase G using the inference rules identified above until saturation or violation detection.

We can show:

Theorem 4.2 (Chase Completeness with Schema) : Let △ be an acylic schema without choice and Q a tree pattern query with NICs but no wildcards. Then Q is satisfiable w.r.t. △ iff there is an embedding of Q into △ and no violation is detected when the constraint graph of Q is chased. ▮

To understand the implications of Theorem 4.2 for the complexity of checking satisfiability of a tree pattern query w.r.t. a given acyclic schema without choice, we consider this problem at two levels. Firstly, let △ be any schema. Then we define the language $\mathsf{SAT}_{\triangle}$={Q | Q is a query & Q is satisfiable w.r.t. △}. We call the complexity of checking this membership *query complexity*, by analogy with the notion of data complexity in [22]. Secondly, we define the language SAT to be SAT={(△, Q) | △ is a schema & Q is a query & Q is satisfiable w.r.t △}. We call the complexity of checking this membership the *combined complexity* of satisfiability checking, by analogy to the well-known notion of combined complexity[22].

We have the following results.

Theorem 4.3 (Query Complexity) : The query complexity of satisfiability checking in the presence of acyclic schema without choice is PTIME.

The idea is that we can apply the inference rules to saturation or until a violation is detected, which is a process that takes polynomial time in the size of query. We can also test whether there is an embedding from the query to the schema in PTIME. This yields a polynomial time algorithm for testing satisfiability in the presence of schema. Efficient implementation, similar to that discussed in Section 3.2.1, is possible. The details are omitted. A final note is that VBCs can be easily incorporated in the same way they were for the schemaless case. Thus, we can test satisfiability in polynomial time in the presence of schema and VBCs.

Theorem 4.4 (Combined Complexity) : The combined complexity of satisfiability in the presence of acylic schema without choice is co-NP-complete.

The complexity comes not directly from the chase, but from violation checking. Figure 13 illustrates the violation checking procedure.

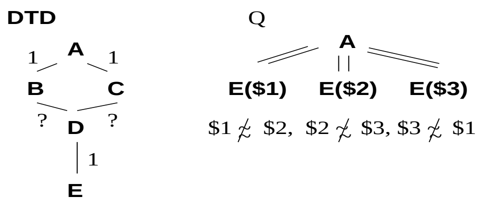


Figure 13: Violation Detection Example

The query Q in Figure 13 is not satisfiable because it asserts there must exist at least three different Es under A. However, there are only two paths from A to E in the DTD, all of whose edges are labeled 1/?. Thus there exist at most two Es under A in any valid instance.

The proof of Theorem 4.4 is by reduction from Maximal Clique. The details can be found in [11]. While the combined complexity is high, in practice, we will often want to check the satisfiability of many queries against a fixed schema, illustrating the significance of query complexity and of Theorem 4.3.

4.2.1 Node Identity Constraints and Wildcards

In the presence of a schema, testing satisfiability of a tree pattern query with wildcards and NICs is NP-complete. Similarly, when there are no wildcards but the query contains value-based disjunctive selection constraints, again the problem is NP-complete.

Theorem 4.5 (Hardness results) : Let $\triangle$ be a schema and Q tree pattern query. Then satisfiability of Q w.r.t. $\triangle$ is NP-complete in the following cases:
(1) Q contains wildcards and NICs.
(2) Q contains disjunctive VBCs (and no NICs).

The first result is by reduction of 3-colorability and the proof only uses $\not\approx$ constraints. The second result is by reduction of 3SAT. Both proofs only make use of tree schemas and only pc-edges in Q.

The complexity results for the schema case are summarized in Figure 14. All results shown correspond to query complexity.

Disjunction	NICs/join constraints	Wildcards	Complexity
		X	PTIME
	X		PTIME
	X	X	NP-complete
X			NP-complete

Figure 14: Complexity of checking Satisfiability with schema.

5 Experimental Results

To study the effectiveness of testing satisfiability, we systematically ran a range of experiments to measure the impact of various parameters. In addition to measuring savings and overhead, we also measured how satisfiability checking time varies as a function of the number and kinds of constraints.

We ran our experiments on the **XMark** benchmark dataset [24] and **Biomedical** dataset [25] from the National Biomedical Research Foundation. For each dataset we constructed the documents of various size using the IBM XMLGenerator [26].

We used *Wutka DTDparser* [27] to parse the DTD, which is needed for static analysis of schema. For query evaluation, we used an XQuery engine *XQEngine* [28] for convenience and flexibility. Both tools are open source, developed in Java. We implemented our satisfiability tests in Java.

Setup: We ran our experiments on a *sparc* workstation running SunOS version 5.9 with 8 processors each having a speed of 900MHz and 32GB of RAM. All values reported are the average of 5 trials after dropping the maximum and minimum, observed during different workloads.

Query Set: All queries chosen for experimentation correspond to classes of tree pattern queries studied in this paper. Please note that when multiple node equalities are present in a TPQ, we need to use XQuery for its implementation.

For satisfiability testing without schema and with schema cases, we used Q1-Q3 in Figure 8. Although

```
Q1:
for $A in doc(‘‘auction.xml’’)//category,
          $B1 in $A//description,
          $C in $A//parlist,
          $B2 in $A//description
   where $B1//text is $C//text and $B2//parlist is $C
   return $A
Q2:
for $A in doc(‘‘auction.xml’’)//description,
          $B in $A/parlist,
          $C in $A//listitem,
          $D in $A//text
   where $B//bold is $C//bold and $D//keyword is $C//keyword
   return $A
Q3:
for $A in doc(‘‘auction.xml’’)//categories
   where $A/description/text is $A//parlist//text
   return $A
```

Figure 15: Examples for Schemaless case

we use the same set of queries, we use different analysis for "no schema" mode and "schema" mode seperately.

We also experimentsed with the **Biomedical** dataset but we did not include the details for space limitations. The details can be found in the full version of this paper.

Saving&Overhead Ratio: Let c be the time taken to determine the satisfiability of a query Q and let e be the time it takes to evaluate the query over the document (without using satisfiability check). The *savings ratio* S_Q obtained by using satisfiability check on unsatisfiable queries is defined as $S_Q = \frac{e-c}{e}$ and the *overhead ratio* incurred by doing satisfiability check on satisfiable queries is defined as $O_Q = \frac{c+e}{e}$. Intuitively, the closer to 1 the two ratios are the better.

Saving Ratio: Not surprisingly, on unsatisfiable queries, satisfiability check leads to phenomenal savings. Our saving ratio is close to 1 (usually between about 0.8 and 0.9) whether the schema is present or not. We omit these results for brevity.

Overhead Ratio: Figures 16 and 17 show the variation of savings ratio with document size for the three satisfiable queries $Q1 - Q3$ in Figure 8 (with schema) and $Q1 - Q3$ in Figure 15 (without schema). We expect the overhead ratio to decrease as the document size increases.

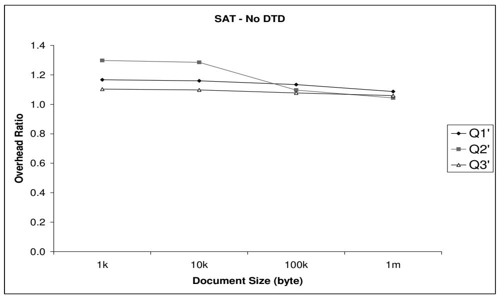


Figure 16: Overhead Ratio - Without Schema

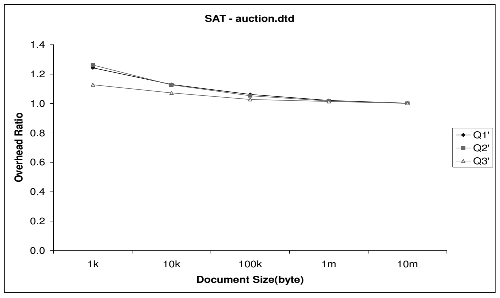


Figure 17: Overhead Ratio - With Schema

Indeed, this behavior can be observed from the figures. Overall, our results show that the overhead is a negligible fraction of the evaluation time.

In addition, we also tested the impact of number of constraints on satisfiability check time. For satisfiable queries, as expected the time increases, while for unsatisfiable queries, it decreases as violations are found faster. We also varied the structure of resulting OTSP sets by adding constraints and studied their effect on satisfiability check time. We found a few large OTSP sets increase the testing time more than several small OTSP sets.

The same conclusions were also obtained from the experiments on the **Biomedical** dataset.

6 Related Work

Containment: There has been much work on query processsing, containment and minimization of various XPath fragments [18, 7, 15, 12, 13, 1, 23]. Kuper et al. [9] study expressive power and closure properties of various XPath fragments and tree pattern queries. Levy et al. [8] studied query equivalence and satisfiability for datalog extensions. Satisfiability can be reduced to containment: query Q is unsatisfiable iff Q is contained in a (fixed) unsatisfiable query Q'. However, our results on satisfiability in this paper cannot be obtained from known results on containment. Specifically, we showed satisfiability can be tested in polynomial time for the following classes of tree pattern queries queries: (i) $\mathsf{TP}^{/,//,[],*}$ and (ii) $\mathsf{TP}^{/,//,[],\mathrm{NIC}}$, both in the absence of a schema and in the presence of an acyclic DTD without disjunction. In the absence of a schema, containment for the former class is co-NP-complete [12] while for the latter it is Π_2^p-complete [18]. While [18] considered containment in the presence of integrity constraints, as pointed out by the authors, they do not capture a DTD completely. Containment for $\mathsf{TP}^{/,//,[],*}$ w.r.t. a DTD was shown to be EXPTIME complete [13], but it should be noted that the DTD is allowed to contain choice and cycles. Complexity of containment when the DTD is acyclic and/or choice-free is open. Finally, complexity of containment for $\mathsf{TP}^{/,//,[],\mathrm{NIC}}$ w.r.t. a DTD is open, although [13] showed that containment for $\mathsf{TP}^{/,[]}$ and $\mathsf{TP}^{//,[]}$ w.r.t. a DTD is co-NP-hard, when the DTD is allowed to contain choice and cycles. In sum, our PTIME results for satisfiability cannot be obtained from known results on containment.

Containment can be reduced to satisfiability: given queries Q, Q', Q is contained in Q' iff $\mathsf{Q} - \mathsf{Q}'$ is unsatisfiable. But this cannot be used to derive the hardness results in this paper since $\mathsf{Q} - \mathsf{Q}'$ does not belong to the class of tree pattern queries studied in this paper.

Satisfiability: The closest work is Hidders [5], where he considers the complexity of satisfiability testing for XPath fragments in the absence of schema. However, there are important differences in the contributions of the two papers, as we explain in detail below. The main contribution of [5] was showing that testing satisfiability of XPath expressions is NP-complete for various XPath fragments: (i) XPath with child and self-or-descendant and intersection, (ii) parent, union, and branching, (iii) root, branching, child, parent, self-or-ancestor. All these results depend on wildcard being present in the query. Secondly, he showed that when only branching (and all forward and backward axes as well as order) are present, satisfiability can be tested in polynomial time. For this, he uses a "tree description graph", which is similar to our constraint graph, except VBCs are not considered. The procedure he adopts for satisfiability testing has a flavor similar to our chase, but the "inference rules" are considerably simpler. The main reason is when set operations (union, intersection) are absent, one cannot express equality. In this case, the inferences become much simpler. He also showed that when all the axes and root are present, but none of the set operations or branching are allowed, again satisfiabilty can be tested in polynomial time. A similar comment applies to inferences in this case.

By contrast, all our PTIME results allow branching. In particular, when the query contains no wildcards but contains VBCs and NICs, we give an efficient polynomial time test for satisfiability. This result does not follow from the results of [5]. Besides, we have extended the techniques and results for testing satisfiability in the presence of schema. To the best of our knowledge, this has not been addressed before. Finally, our NP-completeness results are orthogonal to those in [5]. One exception is Theorem 3.3, which as we mentioned, is an easy corollary of a result in [5].

Testing satisfiability of tree descriptions, based on partial tree descriptions is of considerable interest in computational linguistics [10, 2, 16]. Constraint graphs are one kind of partial tree description. Kutz and Brodirsky [10] recently presented an efficient algorithm that checks the satisfiability of *pure dominance constraints*, which describe unlabeled rooted trees using a partial order. For arbitrary pairs of nodes they specify sets of admissible relative positions in a tree. However, the (pure) dominance constraints are a subset of the structure constraints studied in this paper. Besides, relationships such as OTSP and COUS are not considered there, nor is reasoning in the presence of a schema.

There are also other work related to satisfiability problem. Papakonstantinou et al.[14] studied the inference of DTDs for views of XML data. This paper proposed two extensions that enhance DTD's descriptive power. It mentioned that satisifiability for the views produced by the selective queries in the context of the extended DTD can be checked in PTIME. However, the selective queries are only a subset of the TPQs we discussed in our paper; they didn't allow either wildcards or node equality. Thus the problem of checking the satisfiability of selective queries is equivalent of finding the embedding of the query in our paper.

7 Summary

While there has been considerable work on containment and minimization for various XPath and tree pattern query fragments, the related problem of satisfiability has been largely ignored. We developed a method for testing satisfiability of various classes of tree pattern queries, which are known to be closely related to XPath and XQuery and to be of fundamental importance [9]. We study this problem both with and without a schema (acyclic and choic-free) and identify cases in which it is NP-complete and when it is PTIME. For the latter case, we developed efficient algorithms based on a chase procedure. We complemented our analytical results with an extensive set of experiments. While satisfiability checking can effect substantial savings in query evaluation, our results demonstrate that it incurs negligible overhead over satisfiable queries.

Satisfiability, for larger query classes, in the presence of cycles and/or choice are interesting problems. Satisfiability in the presence of XML schema is an important problem. Results on some of these problems will appear in the full version of this paper.

References

[1] Sihem Amer-Yahia et al. Minimization of tree pattern queries. In *ACM SIGMOD Conference*, 2001.

[2] T. Cornell. On determining the consistency of partial descriptions of trees. In *32nd ACL Conference*, 1994.

[3] D. Draper et. al. Xquery 1.0 and xpath 2.0 formal semantics. Technical report, W3C, 2002.

[4] M. F. Fernandez et. al. Xquery 1.0 and xpath 2.0 data model. Technical report, W3C, 2002.

[5] J. Hidders. Satisfiability of xpath expressions. In *DBPL* 2003.

[6] H. V. Jagadish et. al. Timber: A native xml database. *VLDB Journal*, 2002.

[7] C. Koch and G. Gottlob. Xpath query processing. In *9th International Workshop on Database Programming Languages (DBPL)*, Potsdam, Germany, September 2003.

[8] A.Y. Levy et. al. Equivalence, query-reachability, and satisfiability in datalog extensions. In *ACM PODS Conference*, 1993.

[9] G. M. Kuper et al. Structural properties of xpath fragments. In *ICDT* 2003.

[10] M. Kutz and M. Brodirsky. Pure dominance constraints. In *STACS* 2002.

[11] Laks V.S. Lakshmanan et al. On Testing Satisfiability of Tree Pattern Queries. Tech. Report, Dept. of Computer Science, UBC, March 2004. Available from `http://www.cs.ubc.ca/~laks/papers.html`.

[12] G. Miklau and D. Suciu. Containment and equivalent for an xpath fragment. In *PODS* 2002.

[13] F. Neven and T. Sch. Xpath containment in the presence of disjunction, dtds and variables. ICDT 2003.

[14] Yannis Papakonstantinou et al. DTD Inferencefor Views of XML Data In *ACM PODS Conference* 2000.

[15] R. Pichler et al. The complexity of xpath query evaluation. In *ACM PODS Conference*, 2003.

[16] J. Rogers and K. Vijay-Shanker. Reasoning with descriptions of trees. In *ACL Conference*, 1992.

[17] J. Shanmugasundaram et.al. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB Conference*, 1999.

[18] V. Tannen and A. Deutsch. Containment and integrity constraints for xpath fragments. In *8th KRDB*, 2001.

[19] I.Tatarinov et. al. Storing and querying ordered xml using a relational database system. In *ACM SIGMOD Conference*, 2002.

[20] R. Treinen et al. Dominance constraints: Algorithms and complexity. In *3rd conference on Logical Aspects of Computational Linguistics*, 2001.

[21] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems Volume II: The New Technologies.* Computer Science Press, 1989.

[22] Moshe Vardi. The complexity of relational query languages. In *ACM STOC*, 1982,pp 137-146.

[23] Peter T. Wood. Containment for xpath fragments under dtd constraints. icdt 2003.

[24] XMark: http://monetdb.cwi.nl/xml/.

[25] Biomedical database: http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

[26] IBM XML generator: http://www.alphaworks.ibm.com/tech/xmlgenertor.

[27] Wutka DTD parser:http://www.wutka.com/dtdparser.html.

[28] XQuery: http://xqengine.sourceforge.net.