Satisfiability and relevance for queries over active documents

Serge Abiteboul, Pierre Bourhis, Bogdan Marinoiu

Satisfiability and Relevance for Queries over Active Documents*

Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu
INRIA Saclay – Île-de-France and University Paris Sud
Orsay, France
firstname.lastname@inria.fr

ABSTRACT

Many Web applications are based on dynamic interactions between Web components exchanging flows of information. Such a situation arises for instance in mashup systems [22] or when monitoring distributed autonomous systems [6]. This is a challenging problem that has generated recently a lot of attention; see Web 2.0 [38]. For capturing interactions between Web components, we use active documents interacting with the rest of the world via streams of updates. Their input streams specify updates to the document (in the spirit of RSS feeds), whereas their output streams are defined by queries on the document. In most of the paper, the focus is on input streams where the updates are only insertions, although we do consider also deletions.

We introduce and study two fundamental concepts in this setting, namely, satisfiability and relevance. Some fact is *satisfiable* for an active document and a query if it has a chance to be in the result of the query in some future state. Given an active document and a query, a call in the document is *relevant* if the data brought by this call has a chance to impact the answer to the query. We analyze the complexity of computing satisfiability in our core model (insertions only) and for extensions (e.g., with deletions). We also analyze the complexity of computing relevance in the core model.

Categories and Subject Descriptors

I.7.2 [**XML**]; H.3.4 [**WWW**]; H.2 [**Database management**]; E.2 [**Data structure**]: Trees

General Terms

Algorithms, Languages

Keywords

Active XML, Query satisfiability, Relevance

*This work is partially supported by ANR-06-MDCA-005 grant DocFlow and by ERC Advanced Grant Webdam on Foundation of Web data management.

PODS'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

Many Web applications are based on dynamic interactions between Web components exchanging flows of information. Such a situation arises for instance in mashup systems [22] or when monitoring distributed autonomous systems [6]. This is a challenging problem that has generated recently a lot of attention; see Web 2.0 [38]. For capturing interactions between Web components, we use active documents receiving flows of update requests. We introduce and study two fundamental concepts in this setting, namely, satisfiability for a document and relevance.

The core of the model consists of active documents interacting with the rest of the world via streams of updates.

In this paper, we consider active documents with a set semantics for the children of a node. For the queries, we use tree-pattern queries with joins whose answers are tuples of bindings of the variables in the pattern. Such a document with queries (i.e. views) defined on it is what we call an *axlog widget*. The term axlog results from the marriage between Active XML (AXML for short) [3] and datalog. The input streams of an axlog widget specify updates to the document (in the spirit of RSS feeds). In most of the paper, the focus is on input streams where the updates are only insertions, although we do consider also deletions. An output stream is defined by a query on the document. More precisely, it represents the list of update requests to maintain the view of the query.

Our main contribution is a study of two novel notions for active document, satisfiability and relevance. First, we say that some fact is *satisfiable* for an active document and a query if it has a chance to be in the result of the query in some future state. In the spirit of the evaluation of tree-pattern queries using datalog [25], we show how to evaluate query satisfiability in datalog, so in in the size of the document. Note that the number of satisfiable tuples may be infinite. We use a finite representation based on tuples with variables for the set of satisfiable tuples. To handle these representations, we use the constraint query language CQL [30].

We also study satisfiability (for a document and a query) for extensions of the model. First we introduce typing. We consider typing for both the documents [18, 21] and the data on the input streams [39]. Since we use set semantics, we adapt DTDs to ignore the ordering of siblings. We show how to evaluate satisfiability for documents constrained by unordered-DTDs. Then we consider a number of nonmonotonic features, like deletions, terminating calls, negation in queries. In particular, we see that negation rapidly leads to undecidability of satisfiability. Finally we consider temporal queries, a most useful feature in the context of active documents, e.g., for monitoring. We extend the model with time and show how to evaluate satisfiability building on constraint query languages.

The second key notion we study is relevance. Given an active

document and a query, a call in the document is *relevant* if the data brought by this call has a chance to impact the answer to the query. This is in the spirit of data relevance in MagicSet [11] and lazy-relevance in [1]. Relevance of function calls has also been studied in [17, 2]. Work on view maintenance also discusses relevance of updates, as in [15, 8]. We show how to evaluate relevance in
in the size of the data. The combined complexity is high and the
algorithm is too expensive for pratical purposes. We propose a weaker condition namely *axlog relevance*, that is easier to verify.

The work presented here has been used in the implementation of an engine for axlog widgets. Axlog widgets can be used to support a number of tasks in distributed environments such as the orchestration of Web services. The axlog engine we implemented has been demonstrated in [7] using a supply chain application [31]. It is used in a new version of a P2P monitoring system we also implemented [6]. The engine takes advantage of an optimization algorithm that combines known datalog optimization techniques (e.g., MagicSet and Differential [16]) as well as novel ones. It is notably based on satisfiability and relevance. The optimization technique is described in a companion paper [5]. The model and the notion of axlog-relevance are also presented there, although informally. Typing (Section 4) as well as the nonmonotonic extensions (Section 5) are not studied there. All the results presented here are new.

We want to stress the fact that we consider only unordered trees (set semantics for the children of a node). Results in [36] indicate that the document satisfiability for ordered trees and types specified by DTDs is much more complicated.

The paper is organized as follows. In Section 2, we formalize the model. In Section 3, we study satisfiability. We consider types in Section 4 and other extensions of the model in Section 5. Section 6 is about relevance. The last section is a conclusion. *Due to space limitations, the proofs are omitted. They can be found in [4].*

2. THE MODEL

In this section, we define the data structure (active documents) and the query language (tree-pattern queries) that we study in the remaining of the paper. We introduce here the core model. We will consider a number of extensions in Sections 4 and 5.

We assume the existence of some infinite alphabets $\mathcal{I}$ of node identifiers, $\mathcal{L}$ of labels, $\mathcal{F}$ of (function) call Ids, and $\mathcal{V}$ of variables. We do not distinguish here between XML data, attributes and labels, i.e., our labels are meant to capture these three notions. We use the symbols n, m, p for node identifiers, $a, b, c...$ for labels, $?f, ?g, ?h...$ for call Ids, possibly with sub and superscripts, and $\$x, \$y, \$z...$ for variables. We consider active documents in the style of AXML [3, 10]. Such documents may be viewed as abstractions of XML documents including calls to external resources, e.g., Web services.

D 1 (A). An active document *is a pair* (t, λ) *where (1)* t *is a finite binary relation that is a tree*[1] *with* $nodes(t) \subset \mathcal{I}$ *; (2)* λ *is a labeling function over* $nodes(t)$ *with values in* $\mathcal{L} \cup \mathcal{F}$*; and (3) the root and each node that has a child are labeled by values in* $\mathcal{L}$ *(so only leaves may be labeled by values in* $\mathcal{F}$*). We also impose that: (4) no call Id occurs more than once in an active document.*

A (data) forest *is a finite set of documents and of trees consisting of a single node with a label from* $\mathcal{F}$.

R 1. *In AXML, a* function call node *has children denoting the parameters of the call. We consider in this paper that calls have already been made and a* function call node*, labeled with a call Id, is just a marker to indicate where the results of the call should go, i.e., as siblings of this node. Observe that, therefore, function call nodes do not have children, so there is no nesting of such nodes.*

Four examples of documents are given in Figure 1. The last document presents the members of the Gemo team. Calls to the personnel database feed the document. In a standard database manner, we use in this paper a set semantics for the children of a node. Two active documents (t, λ), (t', λ') are *isomorphic* if they differ in their node identifiers only. In the following, we will consider that all documents are *reduced*, i.e., that they don't include a tree node with two isomorphic subtrees. Clearly, each document can be reduced by eliminating duplicate isomorphic subtrees, and the result is unique up to isomorphism. These notions are lifted to forests in the straightforward manner.

Calls can be seen as subscriptions (to some services) and are meant to receive streams of updates. Trees evolve in time by receiving such update requests from the services called in them. To simplify, we consider first that (1) the incoming flow of updates consists only of insertions, and (2) these flows return data not containing new calls. More precisely, an *insertion* for an active document I is an expression $add(?f, J)$ where $?f$ is a call occurring in I and J a "passive" active document (i.e., it contains no calls). Let I be an active document, $add(?f, J)$ an update of I and n the node of I labeled $?f$. The *result* of applying $add(?f, J)$ to I, denoted $add(?f, J)(I)$, is the active document obtained from I by adding, as a sibling of n, a fresh copy[2] J' of J. For instance, for I, J, K as in Figure 1, $K = add(?f, J)(I)$. The active document obtained from I by applying a sequence ω of inserts is denoted $\omega(I)$. To generalize, we also see an expression $add(?f, \{J_1, ..., J_n\})$ as an update. Applied to some active document I, it has the same effect as the sequence $add(?f, J_1);...; add(?f, J_n)$ of updates (for some ordering of the updates in the set). Observe that the order of application of these updates is irrelevant. This will no longer be true when we consider extensions of the model. Note also that, by definition of active documents, a tree consisting of a single function call node is not a document. This is ruled out because a call may request the insertion of a set of trees, so yield a forest.

Constraint (4) in the definition of active document may seem arbitrary. We justify it next. We can extend the definition of updates to allow the multiple occurrences of a call Id. When such a call returns some data, isomorphic copies of this data are inserted in the document in various places corresponding to occurrences of the call. This introduces some nonregularity (in the sense of regular trees). To see that, consider the set of documents that can be reached from the document $r[a[?f]][b[?f]]$.

The queries considered in our model are tree-pattern queries. Examples are given in Figure 2. The single lines indicate a parent relationship, and the double lines an ancestor relationship. The $\$$-variables may match any label. A variable $\$x$ is requested to be in the result if marked by a "+". The result therefore consists of tuples over the variables marked with "+". Query q_1 is a Boolean query, and the other three queries return binary relations over $\$x$ and $\$y$. We consider only equality joins for now. We will introduce other comparators further on. Formally, we have:

D 2. **(Tree-pattern query)** *A* (tree-pattern) query q *is an expression* $(E_/, E_{//}, \lambda, \pi)$ *where: (i)* $E_/$*,* $E_{//}$ *are finite, disjoint subsets of* $\mathcal{I} \times \mathcal{I}$*, and* $(E_/ \cup E_{//})$ *is a tree; (ii) The labeling function*

[1]The trees that we consider here are unordered and unranked.

[2]The copy is isomorphic to J and its nodes are disjoint from the nodes of I.

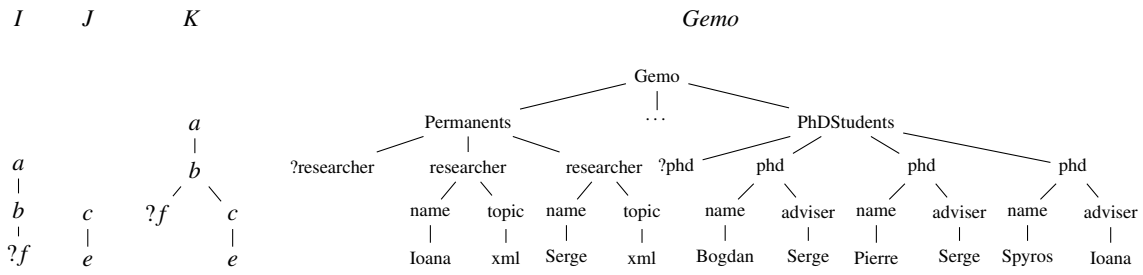

Figure 1: Four examples of active documents

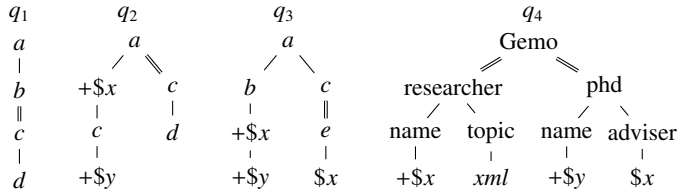

Figure 2: Examples of queries

λ maps nodes(q) to $\mathcal{L} \cup \mathcal{V}$; and (iii) projection π is a subset of nodes(q) with labels in $\mathcal{V}$, where nodes(q) is the set of nodes in $E_/ \cup E_{//}$.

The semantics of queries is defined as follows.

D 3. **(Semantics of TPQ)** *Let $q = (E_/, E_{//}, \lambda, \pi)$ be a query and $I = (t', \lambda')$ a document. A* valuation *ν from q to (t', λ') is a mapping from nodes(q) to nodes(t') that is:*

(i) Root-preserving: $\nu(root(q)) = root(t')$.

(ii) Parent/descendant preserving: For each $(p, p') \in E_/$, $\nu(p)$ is a parent of $\nu(p')$ in t'; and for each $(p, p') \in E_{//}$, $\nu(p)$ is an ancestor of $\nu(p')$ in t'.

(iii) Label-preserving: For each $p \in nodes(q)$, if $\lambda(p) \in \mathcal{L}$ then $\lambda'(\nu(p)) = \lambda(p)$, otherwise $\lambda'(\nu(p)) \in \mathcal{L}$.

(iv) Join-obeying: If $\lambda(p) = \lambda(p') \in \mathcal{V}$, then $\lambda'(\nu(p)) = \lambda'(\nu(p'))$.

The result *$q(I)$ is the relation $\{\lambda'(\nu(\pi)) \mid \nu$ a valuation$\}$.*

If there is no variable occurring more than once, the query is said to be a *no-join* query. If π is empty, the query is said to be a *Boolean* query. Its result is then either the empty set (false) or the set containing the empty tuple (true). For a Boolean query q, if $q(I)$ is true, we say that I *satisfies* q, denoted $I \models q$. For a non-Boolean query q, using standard notation, we denote the fact that a tuple u is a result, i.e. $u \in q(I)$, by $I \models q(u)$.

In general, we can use non-recursive datalog (nrec-datalog for short) to compute the answers to a query in the style of [25, 35]. We assume, in a standard manner, that the document is represented in a relational database using the extensional relations *root, child, descendant, label* with their standard meaning ; in particular, *label*(a, x) holds if the node with identifier x is labeled by $a \in \mathcal{L}$. The representation also uses a unary relation, namely *function*, with the semantics that *function*(x) holds if the label of node x is in $\mathcal{F}$, i.e., x is a function call node. We construct by recursion the datalog program P_q that computes $q(I)$ given I using the programs corresponding to its subqueries. The program has one relation p for each node p of q. The relation for the root of q defines the answer. Observe that the datalog program needs to carry along labels if they are potentially in the result or can potentially be joined to other labels. Observe also that the function call nodes play no role for computing the answers to the query. Details are omitted. Efficient algorithms for evaluation of tree-pattern and XPATH queries can be found in [26].

3. SATISFIABILITY

We are interested in the evolution of *views* over such documents, defined by tree-pattern queries. First, given a document, we want to know if a Boolean query holds in some reachable state. Similarly, we are concerned with determining whether a tuple belongs to the view in some reachable state of the document. These notions are related to that of coverability in dynamic systems [23]. To investigate these issues, we introduce and study the notion of query satisfiability for a document.

Given a document I and a query q, a tuple u is *satisfiable* for (I, q) if $u \in q(\omega(I))$ for some (possibly empty) sequence ω of insertions. We say that a Boolean query q is *satisfiable* for I, if for some ω, $\omega(I) \models q$, i.e., the tuple () is satisfiable for (I, q). This is denoted by $I \models \diamond q$. Clearly, if $I \models q$, then $I \models \diamond q$. In Figure 3, $I_1 \models q_0$, $I_2 \not\models \diamond q_0$ and $I_3 \models \diamond q_0$. For I_3, q_0 does not hold but $?f$ may bring some node labeled c to make q_0 hold. Observe that this leads to some form of 3-valued logic where a tuple may be true, false for now but possibly true in some future, or false forever. This notion is interesting in its own right. For instance, consider in the context of the supply chain application of [31] a query that selects the mail orders that completed successfully. One may want to know which mail orders still have a chance to complete successfully even though they are not part of the query result yet.

We also use a datalog program to compute the set of satisfiable tuples for (I, q). As previously mentioned, in the computation of satisfaction, one carries along the bindings of the variables that may occur in the result or be joined to other labels. For satisfiability, this is more intricate since parts of the bindings may be brought by future inserts and may still be unavailable. In particular, the set of successful bindings for satisfiability may be infinite. (So, there is

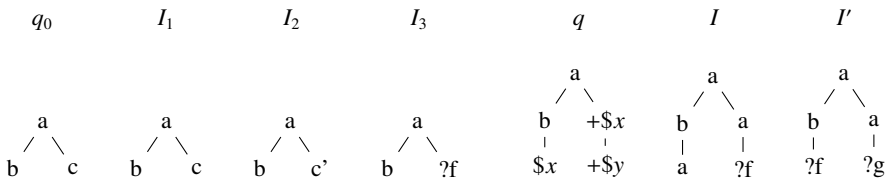

Figure 3: Some queries and active documents

typically no reachable instance that contains all the complete satisfiable tuples since there may be infinitely many such tuples and instances are finite). To overcome this difficulty, we use generalized n-tuples and the constraint query language CQL [30], an extension of datalog with constraints. We only need for now equality constraints between variables and possibly constants.

D 4. *A* generalized n-tuple *is a pair* $(u, \mathcal{C})$ *where* u *is a tuple of variables and* $\mathcal{C}$ *is a set of constraints of the form* $\$x = \$y$ *or* $\$x = a$ *for some a in* $\mathcal{L}$.

A generalized tuple $(u, \mathcal{C})$ is a finite representation for a possibly infinite set of (complete) n-tuples, i.e., the set of tuples $\theta(u)$ for some instantiation θ of the variables satisfying $\mathcal{C}$. We say that a generalized n-tuple $(u, \mathcal{C})$ is *satisfiable* for (I, q) iff for each instantiation θ of u satisfying $\mathcal{C}$, $\theta(u)$ is satisfiable for (I, q). Consider Figure 3. One can verify that, for instance, $(\$x, \$y; \$x = a)$ and $(\$x, \$y; \$x = a, \$y = c)$ are satisfiable for (I, q).

The previous discussion motivates the following auxiliary notion. Let $(u, \mathcal{C})$ and $(u', \mathcal{C}')$ be two generalized n-tuples over the same set of attributes. Then $(u', \mathcal{C}') \sqsubseteq (u, \mathcal{C})$, $(u, \mathcal{C})$ *more general than* $(u', \mathcal{C}')$, iff for each instantiation θ' of u' satisfying $\mathcal{C}'$, there is an instantiation θ of u satisfying $\mathcal{C}$ and $\theta(u) = \theta'(u')$. (This corresponds to the existence of a homomorphism from $(u', \mathcal{C}')$ to $(u, \mathcal{C})$.)

We now sketch the construction of a program P_q that computes the satisfied tuples for an active document. This program is used to build the progam $\widehat{P}_q$ that computes the generalized tuples for satisfiability. Program P_q has two different kinds of intensional relations:

- One intensional relation for each node p, denoted p. The relation p has an arity equal to the number of different variables appearing in the subtree rooted by p plus one.
- One intensional relation q. This is the output relation and has an arity equal to the arity of Π.

A tuple $(n, a_1, \ldots a_k)$ belongs to p iff the tuple $(a_1, \ldots, a_k)$ is an answer of the subquery rooted at p evaluated over the tree rooted at n. Intuitively, the rule associated at the relation p checks if there exists a tuple u belonging to the relation p' for each child p' of p and those tuples satisfy the join constraints over the values and the relations constraints (children and descendants) over the nodes. Due to space limitations, we do not present the construction of P_q.

We now sketch the construction of a program $\widehat{P}_q$ that computes the generalized tuples for satisfiability. It has one relation $\widehat{p}$ for each node p in q with the same arity as p and a relation $\widehat{q}$. Relation $\widehat{p}$ defines the satisfiability for the subquery rooted at p. Program $\widehat{P}_q$ is obtained as follows :

1. Each rule of P_q is rewritten by remplacing each relation p (q) by the relation $\widehat{p}$ $(\widehat{q})$.
2. For each node p of q that is not the root, the following rule is added

$$\widehat{p}(x_1, ..., x_k) \leftarrow function(x_1)$$

where x_1 is the node corresponding to p and $x_2, ..., x_k$ the bindings that are carried are unconstrained along.

Observe es that the second kind of rules may introduce unconstrained variables. This comes from the fact that a call may a-priori bring data matching any pattern. Note also that equality constraints in the generalized tuples are introduced by the joins and the constraints of the generalized tuples are implicit in the rules.

We present here Program $\widehat{P}_q$ in Algorithm 1 associated to the query q of Figure 3. The nodes of the query are numbered using a preorder traversal (the children of the query are ordered for this numbering as in the figure.). So, the node p_1 denotes the root of the query. The program restricted to the six first rules is the part derived from P_q. The four last rules are added to compute the satisfiable tuples in CQL.

Algorithm 1: Program $\widehat{P}_q$ for Query q of Figure 3

begin
1 : $\widehat{q}(x, y) \leftarrow \widehat{p_1}(n, x, y)$
2 : $\widehat{p_1}(n, x, z) \leftarrow root(n), label(a, n), child(n, n'),$
$child(n, n''), \widehat{p_2}(n', x), \widehat{p_4}(n'', y, z), x = y$
3 : $\widehat{p_2}(n, x) \leftarrow child(n, n'), label(b, n), \widehat{p_3}(n', x)$
4 : $\widehat{p_3}(n, x) \leftarrow label(x, n)$
5 : $\widehat{p_4}(n, x, y) \leftarrow label(x, n), child(n, n'), \widehat{p_5}(n', y)$
6 : $\widehat{p_5}(n, y) \leftarrow label(y, n)$
% Satisfiable rules
7 : $\widehat{p_2}(n, x) \leftarrow function(n)$
8 : $\widehat{p_3}(n, x) \leftarrow function(n)$
9 : $\widehat{p_4}(n, x, y) \leftarrow function(n)$
10 : $\widehat{p_5}(n, x) \leftarrow function(n)$
end

In [30], it is shown how to evaluate datalog on generalized tuples in . Using their result, one can show by induction that:

T 1. *Let q be a query. Then there exists an nrec-datalog program* $\widehat{P}_q$ *such that for each generalized n-tuple* $(u, \mathcal{C})$*, (sound) if* $(u, \mathcal{C}) \in \widehat{P}_q(I)$*, then* $(u, \mathcal{C})$ *is satisfiable for* (I, q) *and (complete) if* $(u, \mathcal{C})$ *is satisfiable for* (I, q)*, then there exists* $(u', \mathcal{C}')$ *in* $\widehat{P}_q(I)$*,* $(u, \mathcal{C}) \sqsubseteq (u', \mathcal{C}')$*. Given I, one can compute* $\widehat{P}_q(I)$ *in in the size of the document.*

Observe that the set of tuples returned by the program $\widehat{P}_q$ may be exponential in the size of q. To analyze more precisely the complexity, we turn to Boolean queries. It is interesting to note that a generalized tuple is satisfiable for some (I, q) if the Boolean query $q(\theta(u))$ is satisfiable for I for some θ that maps each variable in u to a distinct new constant not occurring in I or q. We now consider

the complexity of deciding whether a Boolean query is satisfiable for some document.

T 2. *Given I and a Boolean query q, one can decide whether q is satisfiable for I in in the size of I. The problem is - in the size of q (or the size of I and q).*

P . (sketch) The data complexity follows from Theorem 1. NP-hardness is by reduction of the evaluation problem that is known to be - . See Theorem 7.3 of [25]. We now prove that the satisfiability problem is in NP. Consider an instance (I, q) of the problem. To show that $I \models \Diamond q$, it suffices to exhibit a sequence ω of insertions and a valuation ν of q in $\omega(I)$. First, observe that if such a sequence exists, there is one with a number of insertions bounded by $|q|$ and the size of inserted trees also bounded by $|q|$. Furthermore, observe that we need only to consider a polynomial number of labels (we have to guess values for variables). Then we have to check (in polynomial time) that the given *candidate valuation* is successful. So, a polynomial number of guesses (to guess a valuation) followed by a polynomial computation (to check the valuation) suffice. This shows that the problem is in . □

We conclude this section with three remarks: the first is about a subclass of queries for which the problem is easier, and the last two discuss extensions:

R 2 (-). *The program complexity of the problem is - . The complexity comes from the joins. Indeed, one can check whether a Boolean no-join query q is satisfiable for a document I in $O(|q| \times |I|)$. This is based on $\widehat{P}_q$ of Theorem 1 and using [25, 35].*

R 3 ( ). *We can consider insertions of active data (i.e., data including new call Ids). Such feature has no real impact on this section.*

R 4 ( ). *One can consider a system of active documents. In each document, queries produce streams of answers. These streams are used as input streams to other documents of the system (possibly the same document that produced it). Note that as defined, a query produces a set of tuples (answers). These tuples are turned into trees in the obvious way. We assume that some of streams come from the external world (like the input streams we considered so far). Satisfiability of one of the queries in such a system is defined in the obvious way. The datalog computation can be generalized to this setting. Observe that the program may now be recursive. One can show that satisfiability of a query in such a system remains in in the size of the system (i.e., the sum of the sizes of the documents) and that the combined complexity is - . Such systems form the core of the implementation in [5].*

4. TYPED DOCUMENTS

A schema introduces types for the document (as in DTD [21]) and for the return values of calls (as in WSDL [39]). We consider types for unordered unranked trees inspired by DTDs. We study the complexity of satisfiability for queries over documents with schemas specified with such types. Results on query satisfaction (in the classical sense) by static documents constrained by DTDs can be found, e.g, in [12, 19].

DTDs have been defined for unranked ordered trees. We adapt them to our context of unranked *unordered* trees. Recall that we assumed that a call Id occurs at most once in the document. On the other hand, we will accept that a document contains several calls, e.g. $?f_1, ?f_2$ to the same Web service, say w. (For instance, $?f_1$ may correspond to the call $w(0)$ and $f_2?$ to the call $w(1)$.) We assume the existence of an infinite set $\mathcal{W}$ of Web service names. The types we use are based on cardinality constraints on children of nodes. For example, the following "unordered-DTD" Δ_1 defines all trees that have a root labeled a with one b-child, any c-children (the nodes labeled b or c are leaves), and at least one call to some Web service w that returns only nodes labeled c:

d	$root : a$
	$a \rightarrow \{\|b\| = 1 \ \& \ \|c\| \geq 0 \ \& \ \|w\| \geq 1\}$
	$b \rightarrow$
	$c \rightarrow$
call w	$root : c$
	$c \rightarrow$

Formally, a *cardinality constraint* over some set E is a Boolean combination of expressions of the form $|e| \geq k$, for some $e \in E$ and integer k. A multiset M of E is a function from E to $\mathbb{N}$. A multiset M satisfies a cardinality constraint, denoted $M \vdash C$, if by replacing each $|e|$ by $M(e)$ the cardinality constraint is true.

We use a particular symbol, namely *dom*, to represent the set of *data values*, i.e., the elements of $\mathcal{L}$ except those labels occurring in the type definition. An *unordered-DTD* is a an expression (τ, L, W, r) (denoted τ when the other symbols are understood) where L is finite set of labels, W a finite set of Web service names, $r \in L$ is the root label and τ maps each label in L into a cardinality constraint over $L \cup W \cup \{dom\}$. The definition of the satisfaction of an unordered-DTD (τ, L, W, r) by a tree (t, λ) with labels in $\mathcal{L} \cup \mathcal{W}$, denoted $t \models \tau$, is defined as follows: the root must be r; the nodes with labels in $\mathcal{L} - L$ or in W are leaves; the children of each node with label in L must satisfy the corresponding cardinality constraints.

Based on these types, we define schemas. In general, a schema defines the types of severals documents and of severals Web services.

D 5. *A(n* unordered-DTD) schema Δ *is an expression (d, W, ζ) where d is the name of the document, W is a finite set of Webservice names and ζ is a function associating to the document name and to each w in W, an unordered-DTD. An* instance *I of a schema $\Delta = (d, W, \zeta)$ is an expression (t, λ, ν) s.t. the pair (t, λ) is an active document, ν is a function that maps each call Id to a Webservice name and is the identity on $\mathcal{L}$ and the tree $(t, \nu \circ \lambda)$ satisfies $\zeta(d)$.*

Our notion of active document constrained by an unordered-DTD schema is closely related to the notion of incomplete trees of [8].

Satisfiability for unordered-DTD schemas. A schema constrains the evolution of an active document. In particular, some insertions may be inconsistent if they try to transform the instance into an instance not satisfying the schema. For example, consider the unordered-DTD schema Δ_1 previously defined. The insertion $add(?f, c)$ applied to the instance I_3 of Figure 3 gives an instance of Δ_1, by considering that the Id call $?f$ is a call to the Webservice w. But the insertion $add(?f, e)$ leads to a violation of the schema constraint. We assume that an insertion that does not verify the typing constraints is simply rejected. Given a document I satisfying Δ_1, an insertion $\omega = add(?f, K)$ is *valid* if K verifies the signatures of the service corresponding to $?f$ and if $\omega(I)$ verifies Δ_1. A sequence $\omega_1; ...; \omega_k$ of insertions is considered *valid* if for each $1 \leq i \leq k$, ω_i is valid for $\omega_1; ...; \omega_{i-1}(I)$. Observe that for some set of insertions, some sequencing of the set may be valid and some invalid. The document satisfiability problem is extended to take schemas

into account: a query is satisfiable over an instance of a schema (d, W, ζ) iff there is a valid sequence ω of insertions s.t. $\omega(I) \models q$.

The following theorem shows that the document satisfiability problem is still tractable in presence of unordered-DTDs with respect to data complexity.

T 3. *Let a query q and an unordered-DTD schema $\Delta = (d, W, \zeta)$ be fixed. Given I, the satisfiability problem for q, Δ and I an instance of Δ is in in the size of I.*

P . (sketch) To check if a tuple u is satisfiable, we proceed as follows. First the tuple must be satisfiable in absence of typing constraints. We compute the satisfiable tuples and find all those that "cover" u. We prove that u is derived iff there exists a sequence of updates whose length is bounded by a polynomial in the size of the query. Then one has to check, for each call that is performed, that it can bring data matching the desired pattern without violating the typing constraints. Those tests are expensive but do not depend on the data. □

The following theorem shows that the combined complexity remains unchanged in general in the context of DTD's, but goes up for special cases such as no-join Boolean queries.

T 4. *Let an unordered-DTD schema $\Delta = (d, W, \zeta)$ be fixed. Given I and q, the satisfiability problem for q and I constrained by Δ is - in the size of I and q. It is already - for* no-join *Boolean queries.*

P . (sketch) - for no-join Boolean queries is proved by reduction of the satisfiability problem of a no-join Boolean query over a fixed DTD, which is known to be - . See Theorem 4.5 of [12]. Membership in is proved by exhibiting a sequence of updates such that the length of the sequence is bounded by a polynomial in q and the size of each tree is bounded by a polynomial in q. The last property is proved by adapting the proof of Theorem 4.5 in [12], for types on reduced trees. The main technique is to translate an unordered DTD τ to an unordered DTD τ' that "almost" describes the corresponding reduced trees. It captures the trees that can be extended to reduced trees obeying the constraint τ. □

R 5. *When active insertions are allowed, the complexity remains the same, i.e. - in the size of the query and the document and in the size of the document.*

R 6. *The problem becomes undecidable in the case of active insertions when one considers richer typing, namely bottom-up tree automata. Due to space limitations, such richer typing will not to be considered here.*

5. NONMONOTONICITY

We consider in this section nonmonotonic mechanisms: deletions, end of calls, time queries and queries with negation. In this context, satisfiability is no longer monotonic; e.g. a Boolean query may become unsatisfiable during the evolution of document. We consider first mechanisms so that the document is no longer inflationary. We then consider nonmonotonic queries.

Noninflationary documents. We consider two mechanisms that lead to a noninflationary behavior of documents: deletion and end of calls. A *deletion* is a new kind of update of the form $del(?f, q)$ where q is a tree-pattern query to select the nodes to delete. More precisely, the result of applying $del(?f, q)$ to a document I, denoted $del(?f, q)(I)$, is the document obtained by deleting the siblings of node $?f$ satisfying q, as well as their descendants. (In practice, a deletion often uses identifiers to specify the subtrees to be deleted.) We also introduce the possibility that a call terminates. Formally, we consider also messages of the form $eos(?f)$, for end of update stream $?f$. When such a message arrives, the function call node is deleted. Observe that all the operations we consider, insertions, deletions and eos, are in some sense local.

Because of deletions, satisfaction is no longer monotonic. Because of deletions and eos, the document is no longer inflationary and therefore, satisfiability can decrease. One can prove that deletions do not increase the complexity of the satisfiability problem in the simple model. Theorems of Section 3 remain valid in presence of deletions. But, deletions and schema[3] together make the satisfiability problem more difficult. More precisely, Theorems 3 and 4 become :

T 5. *Let a query q and an unordered-DTD schema $\Delta = (d, W, \zeta)$ be fixed. Given I, the satisfiability problem for q and I an instance of Δ and with add, delete updates is - - in the size of I.*

Let an unordered-DTD schema $\Delta = (d, W, \zeta)$ be fixed. Given I and q, the satisfiability problem for q and I constrained by Δ, with add and delete updates is Σ_2^P in the size of I and q.

In our context, we have considered only continuous Web services. In practice, some Web services are one shot, i.e. they send a unique forest as an answer, and terminate. Such one shot Web services do not change much our setting. On the other hand, one may want to impose that the one-shot answer is a single tree, or more generally, that its answer consists of a forest of less than m trees, for some integer m. The satisfiability problem for no-join Boolean queries becomes - when such constraints are imposed even for $m = 1$.

Nonmonotonic queries. We next consider two kinds of nonmonotonic queries: time queries and queries with negation.

Time is present in many examples of queries one wants to ask over active documents. For instance, one may want to detect large-amount mail orders that took more than 2 days to be processed. We next sketch an extension of the model and the query language to support time-based queries relying on systems of inequations. We assume that the definition of an instance I includes a time function ψ from $nodes(I)$ to $\mathbb{Q}$. In general, it would be interesting to also consider data values from $\mathbb{Q}$ and inequations involving data values. To simplify, this is not done here. We impose that in an instance, the time of a node is larger or equal to that of its parent. Furthermore, when applying an update $add(?f, K)$ to an instance I, we impose that (i) the time of each node in K is larger than the time of each node in I; and (ii) the times of all nodes in K are identical. Condition (i) is compulsory to be able to reason about time. Condition (ii) can be relaxed but is used here to simplify.

D 6. *A* time-based query *is a pair (q, C) where q is a query and C is a system of linear inequations over the nodes of q. A* valuation *ν of a query (q, C) in an instance $I = (t, \lambda, \psi)$ is a valuation of q in (t, λ) such that the system of inequations obtained by replacing each node n in C by $\psi(\nu(n))$, is satisfied.*

An example of time-based document I and one of time-based query (q, C) are given in Figure 4. In the graphical representation of documents, we append the time to the label, as in "$a : 2$" for label a and time 2. We use a similar notation in queries.

[3]The schema specifies the nature of updates, inserts or deletes of the funtions occuring in it. Details omitted.

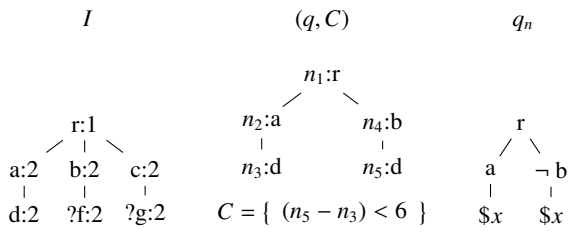

Figure 4: Nonmonotonicity: Time and negation

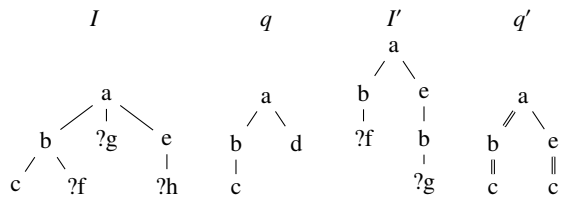

Figure 5: Queries and active documents for relevance

Satisfiability is defined based on this extended notion of valuation in the obvious way. Satisfiability can also be computed in CQL, but the construction is more intricate than previously. We now have to carry along each generalized fact, constraints on its variables. All this can be captured by datalog with constraints [30]. A difficulty is that we don't have the time value of the future data to come. It is important to take into account the fact that this data will have a time larger than the largest time value in the instance (that we can view as the current time). Note that, as a consequence, satisfiability is no longer monotonic. Indeed, the arrival of some data that is seemingly unrelated to the query may turn some query from satisfiable to unsatisfiable simply by updating the current time. To illustrate, consider I and (q, C) in Figure 4. The query is satisfiable for this document. It suffices that $?f$ returns some node labeled d with time say 4. Now suppose that instead, it is $?g$ (seemingly unrelated) that returns some new node with time 10. This is imposing a new constraint on the time of data that will be received later, that makes the query unsatisfiable for the new instance.

T 6. *Satisfiability for time-based documents and queries can be computed by a datalog program with linear inequations as constraints. Thus it can still be tested in in the size of the instance. It is - in the document and query size.*

P . (sketch) To prove the bound, we adapt the program $\widehat{P}$. 3Sat can be reduced to query satisfaction that itself can be reduced to query satisfiability. This shows - . □

To complete the extensions, we consider negation in queries. A tree-pattern with negation is a tree-pattern with an additional unary relation *Neg* over the nodes of the tree-pattern. The meaning of a negation is to state that "there is no subtree matching the pattern". For instance, the Boolean query q_n in Figure 4 states that the r-root has an a-child with a child with some label $\$x$, such that there is no b-child of the root that also has a child labeled $\$x$. In general several negations may be found on a down path from the root. The meaning of the quantification is as follows. Variables not occurring in the output are existentially quantified in the least ancestor of the nodes where they occur. Finally, we also impose that for each variable occurring in the output, at least one of nodes labeled by the variable has all its ancestor non negated.

T 7. *The satisfiability problem is undecidable for queries with negation.*

P . (sketch) The proof is by reduction of the implication problem of functional and inclusion dependencies [29]. This proof is is in the spirit of that of Theorem 4.5 of [8]. Observe that Theorem 7 is more general. □

We observe that in absence of joins, the satisfiability problem is decidable for queries with negation. In the Boolean case, it has data complexity and - combined complexity. It is interesting to obtain restrictions that make satisfiability decidable even in presence of negation and joins. Such restrictions are considered, for instance, in [9].

6. RELEVANCE

We are interested in this section in the possible contributions of call Ids to the result of a query. This problem is particularly useful for optimization. For instance, if we know that a call does not affect the view we want to maintain, we can discard it. We introduce a notion of relevance that captures the intuition that one particular call brings useful information for some particular query we are interested in. We consider here the core model of Section 2 with insertions only and without time. A study of relevance with nonmonotonic features such as deletions is more complicated, and is left for future research.

After some brief motivation, we introduce a semantic notion of relevance and consider its complexity. We then introduce a weaker notion, axlog-relevance, and discuss its completeness and complexity. We briefly mention at the end of the section how we use relevance to optimize axlog widgets.

Intuitively, a call Id $?f$ is "not relevant" for I, q if ignoring the data brought by $?f$ in I does not change the result of q. Consider I and q in Figure 5. Note that $?h$ is not relevant for I and q because its e parent does not match either b or d. Also, $?f$ is not relevant because some sibling already provides the c. On the other hand, $?g$ is clearly relevant for I and q since it can bring a node matching d.

This notion is related to the notion of *lazy-relevance* considered in [1]. They studied the closely related problem: given a query over a document with intentional data (described as Id calls), what are the Web services that need to be called to answer the query. Lazy-relevance can be computed using a tree-pattern query. It is a necessary condition for relevance (as formally introduced further). However, the notions of relevance we consider here are more refined. For instance, the call Id $?f$ in I of Figure 5 is lazy-relevant for I and q, whereas one can see that it is not useful since the only matching data it can bring is a c and we already have one there. We observe in passing that the notion of relevance studied here could be used to improve the query evaluation technique in [1].

To define relevance, we use the following auxiliary notion. Given a sequence ω of insertions and a call Id $?f$, let $\omega_{no\text{-}f}$ denote the sequence obtained from ω by removing all $?f$-insertions. Now, we have:

D 7. *Let q be a query and I an active document. A call Id $?f$ is said to be* not relevant *for q and I iff for each update sequence ω and for each tuple u, $u \in q(\omega(I))$ iff $u \in q(\omega_{no\text{-}f}(I))$.*

This notion of relevance of calls can be carried to data. When the data in a subtree is no longer useful (after all tuples that could be derived using it have been derived), it is not necessary to keep

it. The subtree can then be garbage-collected. This will not be considered here.

The notion of relevance is somewhat more complex that it may look. Indeed, the active document I' and query q' in Figure 5 illustrates a subtlety. Consider the sequence $\omega = add(?f, e[c]); add(?g, c)$, that yields a document satisfying the query. A superficial analysis would lead to believe that $?f$ and $?g$ are both useful because $?f$ can bring data matching the left branch of the query, and $?g$ data for the right branch. However, observe that the update $(?g, c)$ alone is enough to yield a document satisfying the query. Indeed, the update $add(?f, e[c])$ is not needed in that particular sequence and more generally, $?f$ is not relevant for I' and q' as in Figure 5.

Indeed, one can show that:

T 8. *The problem of deciding, given a document and an arbitrary query, whether a call Id is not relevant is in Σ_2^P in the size of the document and the query. The problem of deciding, given a document and a no-join Boolean query, whether a call Id is relevant, is - in the size of the query and the document.*

P . (sketch) Let I be a document containing a call Id $?f$ and q a query. Then $?f$ is relevant for (I, q) if there exists an update ω, such that $\omega(I) = (t', \lambda')$ a tuple u and a valuation θ of the variables in q such that:

(*) for each instanciation θ' from the variables in q to , such that for variable $\$x$ labeling a node of π, $\theta(\$x)) = \theta'(\$x)$

(+) $\omega(I) \models \nu(q)$ and $\omega_{no-f}(I) \not\models \nu'(q)$.

One can show that it suffices to consider ω of polynomial size. Also, the test (+) can be performed in . Thus the problem is in Σ_2^P.

Now consider no-join Boolean queries. - ness is by reduction of 3-SAT. For membership in , let $?f$ be a call Id in a document that is relevant. Then there exists a sequence of updates that demonstrates that $?f$ is relevant. One can show that there also exists a "small" sequence of updates that demonstrates it. So, to compute relevance in , it suffices to guess a "small" sequence of updates and test (*). The test (*) can be performed in for no-join Boolean queries. □

The previous result shows the high expression complexity of the problem. We next show that for any fixed query, relevance can be computed in in the size of the document. To do that, we explore in more details the possible scenarios for obtaining answers to the query, where a scenario is essentially assigning the different roles, i.e. the subqueries to match, to existing data or Id calls occurring in the document. More precisely, we reconsider satisfiability. We extend the generalized tuples used to describe satisfiability by including some "provenance" information. Generally, provenance is used to capture where data *came from*. Here, we are concerned with where data *might come from*. A generalized fact is now of the form $\widehat{p}(u_1, ..., u_n, \mathcal{C}, \mathcal{P})$, where each u_i is some constant or a variable and $u_1, \cdots, u_n$ is a tuple over the variables of subquery p that appear in the result or appear at least twice in q (joins), $\mathcal{C}$ is the set of constraints, and $\mathcal{P}$ is the provenance information defined as follows.

Let $\widehat{p}(u_1, ..., u_n, \mathcal{C}, \mathcal{P})$ be a generalized tuple derived for some query node $\widehat{p}$. The provenance $\mathcal{P}$ is a tuple that specifies how the derivation of corresponding facts depends on the arrival (in certain streams) of data satisfying certain patterns. More precisely, provenance is an m tuple, where m is the number of nodes in the subquery rooted at $\widehat{p}$. The k-th component of $\mathcal{P}$ corresponds to the k-th node of the subquery, in some fixed ordering of these nodes, say preorder traversal. Its value is $\star$ if some data is already present in the document and matches the corresponding query node. It is $n_{?f}$ for some call Id $?f$ if this specific call Id may bring data matching it. It is $\bullet$ otherwise, with the meaning that the data comes from a match in an ancestor node.

We modify the datalog program that computes satisfiability so that it also computes provenance information. We call such tuple (with provenance information), a *scenario*. For the query q and the document I of Figure 5, three tuples are derived (by considering the nodes of the query with the prefix order):

$((), , \star, \star, \star, ?g)$, $((), , \star, \star, ?f, ?g)$, $((), , \star, ?g, \bullet, ?g)$

where the 4 entries of each tuple correspond to provenance for the 4 query nodes in preorder traversal of the query tree. (The query is Boolean, so there is no data to return). Note that each tuple corresponds to a scenario for the possible future derivation of the same fact $q()$. By observing these tuples, one may be led to believe that $?f$ or $?g$ may bring useful data. But since we already obtained the subgoal $a/b/c$, it turns out that this is not the case and only $?g$ is relevant. So, we have to check each scenario to see if it possibly brings new results and in that case, which Id calls are needed.

An instance ω of a given scenario is a sequence of updates, where each update $add(?f, K)$ in ω corresponds to some occurrence of $?f$ in the provenance for a position corresponding to a subquery rooted at p if the edge between p and its parent is a *parent* edge (single line). Furthermore K satisfies the query $\nu(p)$ (p is here the subquery rooted at p) where ν assigns to the result and join variables the values specified by this scenario. (If the edge between p and its parent is a descendant edge, some subtree of K must satisfy it, i.e., double line.)

The algorithm is rather intricate. Its crux is to check (for some I and $?f$) for each scenario where $?f$ occurs, whether there exists a tuple that would be derived in this scenario, and would not have been derived if $?f$ were removed from the scenario. This leads to:

T 9. *Let q be a fixed query. The problem of deciding, given a document and a call Id $?f$ in it whether $?f$ is relevant for I, q, is in in the size of I.*

P . (sketch) Let I be a document and $?f$ a call Id in it. We can compute in all the possible scenarios for (I, q) (i.e., the satisfiable tuples with their provenance). Consider one particular scenario $(u, \mathcal{C}, \mathcal{P})$ including $?f$. (There are polynomially many such scenarios.) Suppose (to simplify the presentation and without loss of generality) that in this scenario $?f$ is used only once. Suppose it is matched to the subquery p. Now consider the query q' obtained from q by pruning out the p subtree. The scenario gives us a scenario for q' that we call the *no-f* scenario of $(u, \mathcal{C}, \mathcal{P})$. To check that $?f$ is relevant for (I, q), it is necessary and sufficient to find a scenario $(u, \mathcal{C}, \mathcal{P})$ and a complete tuple u such that there exists an instance ω of the scenario such that:

(a) the instance ω transforms I into I' with $u \in q(I')$,
(b) the instance ω without $?f$ (which is an instance of the no-f scenario of $(u, \mathcal{C}, \mathcal{P})$) transforms I into I'' with $u \notin q(I'')$.

First observe that, it is possible to restrict our attention to a polynomial number of u tuples. Let u be such a tuple. It is rather easy to test (a). The test of (b) is trickier. Consider the query $\tilde{q}$ obtained by transforming I as follows: the $?f$ call and the $?g$ calls not occurring in the scenario are removed, a $?g$ call occurring in the scenario is replaced by the subquery it is supposed to provide according to this scenario. This query (almost) tests whether a document comes from this particular scenario omitting $?f$. Indeed, one can show that $?f$ is relevant iff there exists an active document J such that $J \models (\tilde{q} \wedge \neg q(u))$, i.e. $\tilde{q} \not\subseteq q(u)$. Intuitively, from such a J, one

can construct an update that demonstrates that $?f$ is relevant. This query containment can be tested in in the size of I. To do that, we eliminate the joins by considering all valuations of the join variables. This results in replacing the containment test by many "easier" containment tests. □

Observe that this technique leads to a lot of computation for each satisfiable tuple. One can avoid testing many of them using a notion of "dominance". For instance, in the previous example, the first tuple dominates the others, so we find immediately that it is the only scenario to consider and we derive that $?f$ is not relevant. This suggest a necessary notion of relevance that we study to conclude this section. It is the one that is used in our system [5].

A *renaming* of a generalized tuple t is a tuple t' obtained by renaming (using a bijection) the variables of t. Let

$$t_1 = (u_1, ..., u_m, \mathcal{C}_1, \mathcal{P}_1),\ t_2 = (u'_1, ..., u'_m, \mathcal{C}_2, \mathcal{P}_2)$$

be two tuples with provenance. (We assume without loss of generality that they have the same "data" part consisting of distinct variables). We say that t_1 is *dominated* by t_2, denoted $t_1 \prec t_2$ if (a) $(u_1, \ldots, u_m; \mathcal{C}_1) \sqsubseteq (u'_1, \ldots, u'_m; \mathcal{C}_2)$ and (b) for each $p \in nodes(q)$, either $\mathcal{P}_2(p) = \star$ or $\mathcal{P}_1(p) = \mathcal{P}_2(p)$ and there exists at least one p such that $\mathcal{P}_2(p) = \star$ and $\mathcal{P}_1(p) \neq \star$. The intuition is that any relevant data needed by the dominating tuple to lead to satisfied tuples is also needed by the dominated one. Thus, the dominated tuples are useless because they lead to the same satisfied tuples.

We refine the set of candidates by eliminating the dominated tuples. In the previous example, the second tuple and the third tuple are eliminated. This leads to the notion of *axlog-relevance*. Let q, I and $?f$ be a query, an active document and a call Id of I. Then $?f$ is *axlog-relevant* for q if there exists a not dominated tuple $p(u, \mathcal{C}, \mathcal{P})$ (i) that may derive new results and (ii) where $?f$ appears. In Figure 5, the first tuple gives a new result so only $?g$ is axlog-relevant. Details omitted.

It is easy to see that axlog-relevance is much more refined than lazy relevance. In particular, in Figure 5 for I and q, the Id call $?f$ is lazy-relevant but not axlog-relevant (neither relevant). On the other hand, axlog-relevance falls short of capturing relevance: In Figure 5, for I' and q', the call Id $?f$ is axlog-relevant but not relevant. To summarize, relevance implies axlog-relevance implies lazy-relevance, and none of the converses hold.

Relevance and axlog-relevance both have complexity in the size of the active document. Axlog-relevance is more tractable in practice since the polynomial has a much smaller coefficient.

We conclude with an observation that turns out to be essential for the optimization of axlog engines:

R 7. *Observe that the computation of satisfiability with provenance information tells us more than just relevance. For a relevant call $?f$, we also find precisely for what it is relevant, i.e., the list of subqueries p_i for which it can bring relevant data. Based on that, we can filter the stream of data brought by $?f$ to let only relevant data enter the document. Indeed, we can even directly feed the relevant data in the relation $\widehat{p_i}$ corresponding to each p_i in the datalog program. In the implementation, we use a YFilter [20] to compute the tuples that are directly fed in the p_i relations. This presents the advantage of reducing processing (fewer data enters the datalog program) and also communication (if the filtering is performed in a remote source).*

7. CONCLUSION

From a technical viewpoint, there are strong connections between our work on satisfiability and the problem of querying an incomplete databases, e.g., [27, 28, 32]; in some sense, the function calls introduce incompleteness. When we consider types, we are very close in spirit to the model of incomplete trees proposed in [8]. In our study of satisfiability, we use previous works on the evaluation of tree pattern queries, notably, [25, 26, 35], and constraint query languages [30].

Our notion of unordered DTD is inspired by other formalisms proposed for unordered unranked trees, e.g., [9]. More general typings have been considered, e.g., [37]. Our results in presence of DTDs use previous results on query satisfiability in presence of DTDs; see, e.g., [12, 14, 19].

The relevance of an update for a query/view has been intensively studied in relational databases, e.g., [15, 33]. In active document contexts, function call relevance has also been studied [1, 2, 8]. The notions of relevance we introduce here (relevance and axlog relevance) are both more refined than previous notions such as lazy relevance. In our study of relevance, we use previous results on query containment [35, 12, 19]. Interesting results of this nature for trees may also be found in [13].

There have been previous works on the verification of temporal properties for active documents. One work [34] also studies active document satisfiability for tree-pattern queries. However, it deals with ordered trees, which is, as mentioned in the introduction, a much more complex issue. The paper is rather imprecise and the results seem to contradict well-known results [36]. [2, 24] study reachability for positive AXML systems (with no deletions). [9] studies a rather general class of non monotone AXML systems and a very large class of temporal formulas. Our model is in many aspects more limited than those used in these previous works. This is the price to pay to obtain data complexities. However, it should be noted that the setting we consider allows unbounded runs (which is not the case in [9]) and infinite data values (which is not the case in [24]).

We mentioned the system that motivated this paper [5, 6, 7]. Satisfiability and relevance are used in the system to optimize change monitoring. Other usages of satisfiability, such as garbage collecting data that becomes useless, may be worth investigating. The work presented here also suggests other directions of research. The introduction of aggregate functions in an active document setting raises interesting issues. In particular, one could consider aggregate functions with time windows, e.g., find the services that are called more than 25 times within 500 time units. Also, our positive results mostly concern the monotone case, i.e., insertions only and monotone queries. It would be interesting to further investigate satisfiability for queries with negation and updates including deletions. This could lead to studying tractable cases for the model of [9]. Finally, it would interesting to investigate tractable cases for ordered trees, possibly based on the decidable cases of [36].

Acknoledgements.

We wish to thank Victor Vianu, Claire David and Evgeny Kharlamov for discussions on this paper.

8. REFERENCES

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *SIGMOD Conference*, pages 227–238, 2004.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *PODS*, pages 35–45, 2004.

[3] S. Abiteboul, O. Benjelloun, and T. Milo. The active XML project: an overview. *VLDB J.*, 2008.

[4] S. Abiteboul, P. Bourhis, and B. Marinoiu. Satifiability and relevance for queries over active documents (full version). ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-10019.pdf.

[5] S. Abiteboul, P. Bourhis, and B. Marinoiu. Efficient maintenance techniques for views over active documents. In *EDBT*, 2009.

[6] S. Abiteboul and B. Marinoiu. Distributed monitoring of peer to peer systems. In *Workshop On Web Information And Data Management*, pages 41–48, 2007.

[7] S. Abiteboul, B. Marinoiu, and P. Bourhis. Distributed Monitoring of Peer to Peer Systems (demo). In *ICDE*, 2008.

[8] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying xml with incomplete information. *ACM Trans. Database Syst.*, 31(1):208–254, 2006.

[9] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *PODS*, pages 221–230, 2008.

[10] Active XML. http://activexml.net.

[11] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3-4):255–299, 1991.

[12] M. Benedikt, W. Fan, and F. Geerts. Xpath satisfiability in the presence of dtds. In *PODS '05*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[13] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. In *DBPL*, pages 66–80, 2007.

[14] H. Björklund, W. Martens, and T. Schwentick. Optimizing conjunctive queries over trees using schema information. In *MFCS*, pages 132–143, 2008.

[15] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *VLDB'86*, pages 457–466, 1986.

[16] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, 1986.

[17] A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE*, pages 50–59, 2008.

[18] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[19] C. David. Complexity of data tree patterns over xml documents. In *MFCS*, 2008.

[20] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–, 2002.

[21] DTD. http://www.w3.org/tr/rec-xml/#dt-doctype.

[22] R. Ennals and D. Gay. User-friendly functional programming for Web mashups. In *ICFP*, pages 223–234, 2007.

[23] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, Apr. 2001.

[24] B. Genest, A. Muscholl, O. Serre, and M. Zeitoun. Tree pattern rewriting systems. In *ATVA*, pages 332–346, 2008.

[25] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *LICS*, pages 189–202, 2002.

[26] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[27] G. Grahne. *Problem of Incomplete Information in Relational Databases*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[28] T. Imielinski and J. W. Lipski. The relational model of data and cylindrical algebras. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 170–170, New York, NY, USA, 1982. ACM.

[29] C. A. K. and V. M. Y. The implication problem for functional and inclusion dependencies is undecidable. *SIAM journal on computing*, 14(3):pp. 671–677, 1985.

[30] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995.

[31] R. Kapuscinski, R. Q. Zhang, P. Carbonneau, R. Moore, and B. Reeves. Inventory decisions in Dell's supply chain. *Interfaces*, 34(3):191–205, 2004.

[32] A. Y. Levy. Obtaining complete answers from incomplete databases. In *In Proc. of the 22nd Int. Conf. on Very Large Data Bases (VLDB'96*, pages 402–412, 1996.

[33] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 171–181, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[34] A.-T. Ma, Z.-X. Hao, and Y. Zhu. Checking satisfiability of tree pattern queries for active xml documents. In *INFOCOMP*, pages 11–18, 2008.

[35] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[36] A. Muscholl, T. Schwentick, and L. Segoufin. Active context-free games. In *STACS*, pages 452–464, 2004.

[37] H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 155–166, New York, NY, USA, 2003. ACM.

[38] What Is Web 2.0. http://www.oreilly.com/.

[39] WSDL. http://www.w3.org/tr/wsdl.