Active documents: satisfiability of queries and view maintenance

Serge Abiteboul Pierre Bourhis* Bogdan Marinoiu
INRIA Orsay† and University Paris Sud
firstname.lastname@inria.fr

ABSTRACT

We consider the evolution of active documents, and, more precisely, of Active XML documents with incoming streams of data. A main contribution of the paper is a study of "document satisfiability" defined as follows. A formula is *satisfiable for a document* if it may hold in some future (depending on data brought by the incoming streams). We show how to evaluate document satisfiability of tree-pattern queries using non-recursive datalog. We characterize the complexity of this problem for important variants of the document model and the query language. We use this notion together with known datalog techniques (Differential and MagicSet) for maintaining views over active documents. We also exhibit optimization techniques that consist in pushing filters to the incoming streams and garbage collecting data and streams that are useless with respect to the views of interest.

Keywords.

XML, active XML, active document, incremental evaluation, tree-pattern query, view maintenance, datalog, monitoring, stream processing

1. INTRODUCTION

In this paper, the focus is on active documents, more precisely, on XML documents that evolve in time because of the presence of incoming input streams. We introduce the novel notion of *document satisfiability* defined as follow. An active document is *satisfiable* for a formula φ if there exists a sequence of inputs that transforms the document into one that satisfies φ. A main contribution of the paper is a complexity analysis for document satisfiability. Based on document satisfiability, we show how to use existing datalog optimization techniques (MagicSet and Differential) to optimize the view maintenance of active documents. We introduce novel optimization techniques for view maintenance that are also based on this notion.

Our work is in the context of Active XML (AXML for short) documents [2], i.e. XML documents with embedded service calls. We restrict our attention to unordered documents, with only insert messages. For queries, we use tree-pattern queries with join. We also study extensions of the document model and of the query language (notably with time).

Consider a monotone formula posed to an active document that evolves in time by receiving incoming flows of data. The content of the document keeps increasing because of the incoming data. At some particular time, three cases can arise: (i) the formula holds, (ii) it does not but has a chance to hold in the future, and (iii) it does not and has no chance to hold in the future. This is some form of 3-valued logic. Now consider a monotone query. Some tuple that is not part of the result at the present time, may have a chance to be derived in the future. To represent the (possibly infinite) set of tuples that have a chance to be derived, we use incomplete information. We show how to compute such a representation using non-recursive datalog, so that it can be computed in *PTIME* in the size of the document. We show that the combined complexity of checking whether a Boolean query is satisfiable for a document is *NP-COMPLETE*. We study important restrictions and extensions of the document model (e.g., based on typing) and the query language (e.g., negation).

The notion of satisfiability is interesting in its own right, e.g., a user may want to ask if a query has a chance to succeed. It turns out to also be useful for maintaining views of active documents. Such view maintenance is important in practice. For instance, one can represent some activity using an active document as a *business artifact* [23]. The surveillance of the activity can then be performed as the maintenance of a view of the document. The issue of view maintenance of active documents also arises in the context of the monitoring of distributed systems. The monitored systems generate XML streams of alerts. Monitoring queries can be specified as queries over active documents including these streams (as input) [4].

As already mentioned, we show how to use non-recursive datalog to maintain incrementally views over active documents. We optimize view maintenance using known datalog optimization techniques, namely MagicSet and Differential. As we will see, using document satisfiability (instead of the classical notion of satisfaction), one can be more aggressive in deriving relevant tuples, so spread more evenly the view maintenance load. Finally, we use document satisfiability in novel optimization techniques for view maintenance, adapted to our context. We show how to push filters to the input streams, which is particularly useful for streams on distant machines. We also introduce other specific optimization techniques, and in particular, one allowing to garbage collect portions of active documents (possibly streams) that are useless with respect to the views to maintain.

*ENS Cachan.
†Address: Parc Club Orsay Université, ZAC des vignes, 4, rue Jacques Monod - Bâtiment G, 91893 ORSAY Cedex

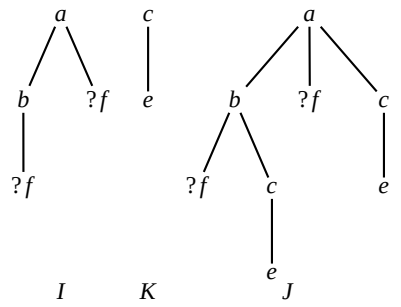


Figure 1: Examples of documents

Due to space limitations, only sketches of proofs are given in the paper. Details can be found in an appendix. The techniques described here are used in a system, called P2P Monitor [4] for monitoring XML streams. To illustrate the use of our results, this implementation is briefly discussed. In particular, we briefly discuss how the techniques can be implemented using XQuery queries instead of datalog, as done in the actual implementation. Datalog is used in the paper (i) to facilitate the presentation and (ii) to be able to reuse the rich technology developed around datalog for query optimization, view maintenance and constraint queries.

The paper is organized as follows. The model is presented in Section 2. Satisfiability is considered for the core model in Section 3 and for extensions in Section 4. View maintenance is the topic of Section 5 and further optimization techniques that of Section 6. Next, an implementation is discussed in Section 7. The last section is a conclusion.

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. The model we introduce is limited. We will consider a number of extensions in Section 4.

We assume the existence of some infinite alphabets $\mathcal{I}$ of node identifiers, $\mathcal{L}$ of labels, $\mathcal{F}$ of function names and $\mathcal{V}$ of variables. We do not distinguish here between 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 function names, d for document names, possibly with sub and superscripts, and $\$1, \$2...$ for variables.

We consider active documents in the style of AXML [2, 7].

DEFINITION 1. **(Active document)** *An active document is*[1] *a pair* (t, λ) *where (1)* t *is a finite binary relation that is a finite tree*[2] *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* $\mathcal{F}$*).* A (data) forest *is a finite set of documents and of trees consisting of a single node with a label from* $\mathcal{F}$.

Three examples of documents are given in Figure 1.

In the previous definition, observe that a tree consisting in a single node labeled by a function is not a document. This is because a function may return a forest.

[1]Our definition is a restriction of AXML. In particular, we assume here that the calls have already been activated, so the function symbols are simply place holders for receiving results of the corresponding calls.

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

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.

To simplify the presentation, we consider in this paper that schemas consist of a single document. Formally, a *schema* is a pair (d, F) where d is a document name and F a finite set of function names. An *instance* I of (d, F) is an active document with only function names in F.

If $\lambda(n)$ is in $\mathcal{L}$, the node is called a *data node* whereas if it is in $\mathcal{F}$, it is called a *function node* (service call in AXML terminology). Function nodes can be seen as subscriptions (to some services) and are meant to receive data updates. Trees evolve in time by receiving such update requests from the functions that occur in them. In the present paper, unless specified, we consider that the incoming flow of updates consists only of insertions. As a consequence, the content of the document is monotonously increasing. More precisely, an *update* for a document (d, F) is an expression $add(?f, K)$ where $?f \in F$ and K is a data tree with functions only in F. Let I be an instance of (d, F) and $add(?f, K)$ an update of I. The *result* of applying $add(?f, K)$ to I, denoted $add(?f, K)(I)$, is the instance obtained from I by adding, for each node n labeled $?f$ in I, a fresh copy[3] $h_n(K)$ of K, as a sibling of n. For instance, for I, K, J as in Figure 1, $J = add(?f, K)(I)$. The instance obtained from I by applying a sequence ω of updates is denoted $\omega(I)$.

To generalize, we see an expression $add(?f, \{K_1, ..., K_n\})$ also as an update. Applied to some instance I, it has the same effect as the sequence $add(?f, K_1)...add(?f, K_n)$ of updates. Observe that the order of application of these updates is irrelevant.

To simplify the presentation, unless specified, the functions we consider here return data without function symbols, i.e., passive data.

The fact that a function occurs more than once is introducing some nonregularity (in the sense of regular trees). To see that, consider the set of documents that can be reached from the document $r[a[?f]][b[?f]]$. Even if we assume that $?f$ returns only passive data, and if we consider only documents over a finite set of labels, the resulting set of documents cannot be described by an unranked tree automaton [11]. Due to space limitations, the details are omitted. This digression was meant to stress the distinction between instances with or without repeated function names.

We consider here tree-pattern queries. Examples of queries are given in Figure 2. The single lines indicate a parent relationship, and the double lines an ancestor relationship. The dollar-variables may match any label. A label is requested to be in the result if marked by a "+". Query q_1 is a Boolean query, q_2 returns a binary relation of labels and q_3 a set of labels.

Considering only equality joins to simplify, we formally have:

DEFINITION 2. **(Tree-pattern query)** *A* (tree-pattern) query q *is an expression* $(E_/, E_{//}, \lambda, \pi)$ *where:*

- $E_/$, $E_{//}$ *are finite, disjoint subsets of* $\mathcal{I} \times \mathcal{I}$*, and* $(E_/ \cup E_{//})$ *is a tree;*
- *The labeling function* λ *maps nodes*(q) *to* $\mathcal{L} \cup \mathcal{V}$*; and*
- *The projection* π *is a subset of nodes*(q)*;*

[3]Each copy of K that is inserted is an active tree isomorphic to K with pairwise disjoint nodes and nodes disjoint from the nodes of I.

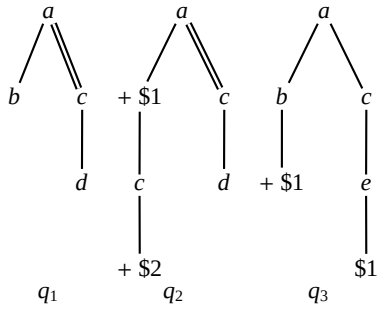


Figure 2: Examples of queries

where nodes(q) *is the set of nodes in* $E_/ \cup E_{//}$.

As a standard notation, we sometimes use "*" to denote a label variable that occurs only once in the tree-pattern and not in the output (π).

The semantics of queries is defined as follows.

DEFINITION 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:*

- *Root-preserving:* $\nu(root(q)) = root(t')$.
- *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'.
- *Label-preserving: For each* $p \in nodes(q)$, *if* $\lambda(p) \in \mathcal{L}$ *then* $\lambda'(\nu(p)) = \lambda(p)$, *otherwise* $\lambda'(\nu(p)) \in \mathcal{L}$.
- *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 $I \models q(u)$.

3. SATISFIABILITY FOR A DOCUMENT

The documents evolve in time. So, a query that is not satisfied at some particular time, may become satisfied in the future. This motivates the following definition, in a temporal logic style. A Boolean query q is *satisfiable* for some active document I if there exists a (possibly empty) sequence ω of updates such that $\omega(I) \models q$. This is denoted by $I \models \diamond q$. Clearly, if $I \models q$, then $I \models \diamond q$. In Figure 3, $I_1 \models q$, $I_2 \not\models \diamond q$ and $I_3 \models \diamond q$. For the last one, q does not hold but f may bring some node labeled c to make q hold.

One main contribution of this paper is a study of the satisfiability for a document by a query. More precisely, we consider the complexity of the *document satisfiability* problem for the core model in this section, and for extensions in Section 4.

Query satisfaction (for queries considered here) can be checked in non-recursive (nrec for short) datalog, so is in PTIME in the size of the data [16, 22]. We next show that nrec datalog can also be used to compute the set of tuples that are satisfiable for a document.

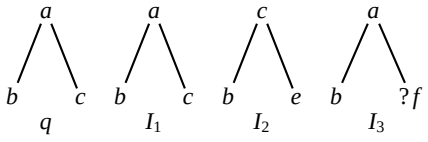


Figure 3: A query and three documents

For this, we assume in a standard manner, that the document is represented in a relational database using the extensional relations *root, child, descendant, label, function* with the obvious meanings ($label(a, x)$ holds if the node with identifier x is labeled by $a \in \mathcal{L}$). Now we have:

THEOREM 1. *Let* q *be a no-join Boolean query. There exists a monadic nrec-datalog program* δ_q, *such that* q *is satisfiable for some document* I *iff* $\delta_q(I)$ *is not empty. For such queries, document satisfiability can be tested in* $O(|q| \times |I|)$, *so in particular, is in* LINEAR *time in the size of the instance.*

PROOF. (sketch) For such queries, satisfiability for a document can be solved recursively by considering the condition on the root and the subconditions on the subtrees. By recursion, one can construct a monadic nrec-datalog program that computes satisfiability. This program can be constructed in LINEAR time in q and after rewriting, there is a monadic datalog program (without using descendant relation) in a size linear of q, that can be evaluated in LINEAR time in I (see [16]).

Some key lemma for this proof and the algorithm are given in appendix. □

Now consider more general queries. We have to carry along the return values. Similarly, we have to carry along labels until they are joined with other labels. In the classical case of satisfaction, one can essentially carry the bindings for all the query nodes, then perform the joins and the appropriate projection to those nodes that are in the result. For document satisfiability, this is more intricate since part of the bindings may be brought by future updates and may still be unavailable. In particular, the set of successful bindings may be infinite. To overcome this difficulty, we use incomplete tuples.

Observe that a valuation in our context is a tuple over the nodes in the query tree. An *incomplete tuple* is a tuple with values in $\mathcal{L} \cup \mathcal{V}$. For an incomplete tuple u and a query q, we say that $I \models \diamond q(u)$ if for each *instantiation* θ of the variables, $I \models \diamond q(\theta(u))$ (where an instantiation of an incomplete tuple u is a function mapping the variables in u to labels in $\mathcal{L}$). In other words, $q(u)$ is seen as the formula obtained from $q(u)$ by quantifying universally all the variables in u. Consider the example of Figure 4. One can verify that $I \models \diamond q(a, \$3)$.

The previous definition motivates the following auxiliary notion. Let u and u' be two incomplete tuples over the same set of attributes. Then $u \Rightarrow u'$ (u more general than u') if for each n, n' in $\mathcal{I}$, if $u(n) \in \mathcal{L}$ then $u'(n) = u(n)$ and if $u(n) = u(n')$ then $u'(n) = u'(n')$. (Observe that this corresponds to quantifying universally all the variables in incomplete tuples.)

THEOREM 2. *Let* q *be a query. Then there exists a nrec-datalog program* δ_q *such that for each incomplete tuple* u, *(sound) if* $u \in \delta_q(I)$, *then* $I \models \diamond q(u)$ *and (complete) if* $I \models \diamond q(u)$, *then there exists* u' *in* $\delta_q(I)$, $u' \Rightarrow u$.

Thus, satisfiability for a document can be tested in PTIME *in the size of the document.*

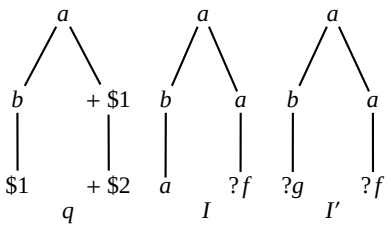


Figure 4: A query and two documents

PROOF. (sketch) The construction is again by recursion. Note that we have to process incomplete tuples, and in particular, to use unification when joining them. The data complexity follows as in datalog, because (i) each step can be achieved in PTIME and (ii) there are polynomially many steps. See appendix. □

Observe that the set of maximal tuples satisfying $\diamond q$ (up to equivalence) may be exponential in the size of q. To conclude this section, we consider the combined complexity of the document satisfiability problem, i.e., the complexity of deciding given I, q and some incomplete tuple u whether $I \models \diamond q(u)$.

THEOREM 3. *Document satisfiability by an incomplete tuple is* NP-COMPLETE *in the size of the instance and the query.*

PROOF. NP-hardness is by reduction of the satisfaction problem that is known to be NP-COMPLETE. See Theorem 7.3 of [16] .

We now prove that the satisfiability problem is in NP. Consider an instance of the problem consisting of I, q and some incomplete tuple u. To show that $I \models \diamond q(u)$, it suffices to show that for each instantiation θ of u, $I \models \diamond q(\theta(u))$. In fact, one can show that it is sufficient that $I \models \diamond q(\theta(u))$ for one particular instantiation θ that maps each variable in u to some new fresh label. See Lemma 4 in appendix. This reduces the problem to that of the satisfiability for the complete tuple $\theta(u)$.

Now consider $q' = q(\theta(u))$, a Boolean query. To show that $I \models \diamond q'$, it suffices to exhibit a sequence of updates ω 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 a 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 NP. □

4. EXTENSIONS

In this section, we consider extensions of the document model. We then consider monotone extensions of the query language and finally non monotonic features.

Active results and terminating functions.

One can consider functions possibly returning active data (trees with function calls) and end-of-stream messages. These two features do not change anything from the document satisfiability viewpoint. (On the other hand, we will see that they have an impact on the problem we study further, incremental query evaluation.)

We next consider 4 other extensions of the document model. We first consider two kinds of type restrictions, on the functions and on the documents. For types, we use *no-order-DTDs*, i.e., DTDs ignoring a number of features absent from our model, in particular the ordering of siblings. A no-order-DTDs restricts for each label a, the labels of children that a-nodes may have. As our trees are unordered, we use Boolean combinations of statements of the form $|b| \geq k$ for a label b and a non-negative integer k. Validity with respect to a no-order-DTD is defined in the standard way. Details are omitted.

Types interact with the reducibility of documents in subtle ways as shown by the following example. Consider a no-order-DTD Δ specified by:

$$\begin{array}{lll} a & \rightarrow & \{|b| \geq 2\} \\ b & \rightarrow & \end{array}$$

Observe that the unreduced document $a[b, b]$ verifies Δ, but no reduced one does.

We next consider the typing of functions and documents. In each case, we show that the introduction of typing increases the combined complexity of the problem. We believe that the data complexity remains PTIME. This would require some in-depth study of no-order-DTD and is left for future works.

Typed documents.

We may want to impose type restrictions on documents. Such typing increases the complexity of the problem. In particular, in contrast with Theorem 1, we have:

THEOREM 4. *The document satisfiability problem of no-join Boolean queries for a document typed by a no-order-DTD is* NP-HARD *in the size of the query and the size of the type.*

PROOF. (sketch) The proof of hardness is by reduction of 3SAT [14]. Let $\varphi = \wedge_{i\in[1..n]} C_i$ be a 3SAT formula. We assume without loss of generality that no clause contains a variable x_j and its negation $\bar{x_j}$. From φ, we construct an instance of the document satisfiability problem, i.e., a document I_φ and a query q_φ, as follows. For each variable x, let l_x be a distinct label. For each C_i, let c_i be also a distinct new label. The document uses some function $?h_x$ for each variable x. The active document I_φ is as follows: the root, labeled r, has one subtree t_x for each variable x. The subtree t_x has a root labeled l_x and a node labeled $?h_x$ as a single child. The query q_φ has its root labeled r with n subtrees all with root labeled $*$ and with a single child labeled c_i, $1 \leq i \leq n$.

The type of d is constrained by imposing that the children of a node labeled l_x for each variable x are: $?h_x$ and either some of the clauses where x occurs positively or some of the ones where it occurs negatively. This is a no-order-DTD constraint.

One can show that q_φ is satisfiable for I_φ iff φ is satisfiable. □

Typed functions.

One may want to impose type restrictions on the return values of functions. We start by illustrating that typed functions increase the complexity. Consider a document I consisting of a root labeled r with a function $?f$ as single child and a query q consisting of a root r with as single subtree the Boolean query q_1. Suppose we know that all results returned by $?f$ satisfy a query q_2. Then q is satisfiable for I if $q_1 \wedge q_2$ is satisfiable (in the classical sense that there is a document satisfying it). Depending on the type language and the query language, this may be expensive to check [9]. Indeed, in contrast with Theorem 1, we have:

THEOREM 5. *The satisfiability problem for documents with functions typed by no-order-DTDs and no-join Boolean queries is* NP-HARD *in the size of the query and the size of the type.*

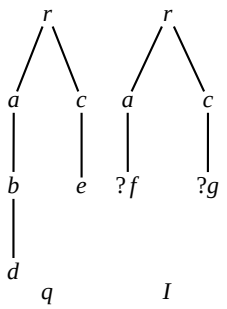


Figure 5: Query and document for the scenario example

PROOF. The proof of NP-HARDNESS is in the style of the one for Theorem 4. □

We next turn to the third extension of the document model.

Non stream functions.

A *non-stream* function is a function that returns a tree and terminates (i.e., it is a function in the classical sense, whereas the functions we usually consider in the paper are stream functions.) Such functions increase the complexity of the problem. In particular, in contrast with Theorem 1, we have:

THEOREM 6. *The satisfiability problem for documents with* non-stream *functions and no-join Boolean queries is* NP-COMPLETE *in the size of the query.*

The satisfiability problem for documents with non-stream *functions and no-join Boolean queries is in* PTIME *in the size of the document.*

The proof uses the auxiliary notion of *scenario* that turns out to be an essential tool in our context. For the satisfiability of a query, we are interested in some function node n bringing some data that matches some query node p. A *scenario* is a tuple u over $nodes(q)$ with values in the set of function names that appear in I union $\{\perp\}$. The meaning of $u(p) = ?g$ is that, in this scenario, $?g$ brings data that matches the subquery rooted at p. Let us denote p' the parent of p in the query tree. If $(p', p) \in E_{/}$, the subquery rooted at p has to match the tree t brought by $?g$ starting with the root of that tree. Otherwise,$(p', p) \in E_{//}$ and the subquery has to match starting with some node in t. The meaning of the $u(p) = \perp$ is that, in this scenario, no data for the corresponding subquery is considered.

A scenario u derived for the query q and the document I of Figure 5 is as follow: $u(b) = ?f$, $u(e) = ?g$ and $u(a) = u(c) = u(d) = \perp$ (each query node is denoted here by its label since there is no ambiguity). It is the only successful scenario.

PROOF. (sketch) The proof of the NP-HARDness is in the style of the one for Theorem 4 and is omitted.

NP (combined complexity): To show that q is satisfiable for d, it suffices to exhibit an update sequence ω such that $\omega(I)(d) \models q$ and that the update sequence contains only one message per function. The test can be performed in PTIME. It remains to see that it suffices to consider a sequence of polynomial size in $|q|$. Suppose such an ω exists. We can take a minimum subsequence of ω so that $\omega(I)(d) \models q$. By definition of q, there is one, say ω', that has no more updates than $|q|$. Now it could be the case that the size of one update in ω' is very large. But one can show that its "useful" part is not.

PTIME (data complexity): We build an algorithm that determines all the *scenarios* that reach query satisfaction. The number of these scenarios is exponential in the query size but polynomial in the document size. For each scenario, it suffices to test that one message per function is enough to make the scenario viable. It suffices that one scenario passes the test to make the query satisfiable.

Details may be found in appendix. □

Finally, we consider the fourth and last extension of the document model.

Time-based queries.

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 older than 3 days that have not been processed yet. We sketch next 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 τ 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.

DEFINITION 4. *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* $I = (t, \lambda, \tau)$ *is a valuation of* q *in* (t, λ) *such that the system of inequations obtained by replacing each node* n *in* C *by* $\tau(\nu(n))$*, is satisfied.*

An example of time-based document and one of time-based query are given in Figure 6. In 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.

Document satisfiability is defined based on this extended notion of valuation in the obvious way. We can compute document satisfiability using datalog as before. We now have to carry along each incomplete tuple, some constraint on the variables of the tuple, a system of inequations. So, this leads to datalog with constraints [21]. A difficulty is that we don't have the time value of the future data to come. It is important to take into consideration 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). Indeed, it is easy to update this current time by having a "clock" that regularly sends in some dummy data whose sole purpose is to impose some new minimal time basis for the data that will come.

Observe that document satisfiability is no more monotone. 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 the example of Figure 6. The query is satisfiable for this document. It suffices that $?f$ returns some node labeled d with time say 4. Now suppose that it is $?g$ (seemingly unrelated) that returns some new node with time 10. This is imposing new constraint on the time of data that will be received later on, that makes the query unsatisfiable for the new instance.

THEOREM 7. *Satisfiability for time-based documents and queries can be verified by a datalog program with linear inequations as constraints. Thus it can still be tested in* PTIME *in the size of the instance.*

It is NP-HARD *in the document and query size.*

PROOF. We prove the PTIME complexity in the data by building a program in Constraint Query Language (CQL) [21] that evaluates the satisfiable incomplete tuples.

The first step is to adapt the datalog algorithm for evaluating the satisfiability of queries to the case with constraints, by considering generalized tuples.

In a second step, we augment the relations p with attributes corresponding to times of nodes. Actually, the idea is to propagate the times for nodes that map those pattern nodes which appear in the time constraints. Generalized tuples of a relation p are of the form (n, u, v_t), where (n, u) is the tuple in the initial datalog algorithm and v_t the times of nodes in the constraints. These time constraints will all be taken into account by a last datalog rule, at the level of the pattern root node.

In a third step, we modify the rules that map the functions to pattern nodes. Intuitivelly, the nodes that are brought by an update have to respect two additional conditions to be mapped to a relation p:

1. The times of nodes brought by the update are greater than the current time (that is given in an extended relation).
2. The times of nodes brought by the update are all equal. (Condition (ii) previously introduced.)

Observe that our framework imposes that if an update u' arrives after an update u, the nodes brought by u' have a time greater or equal to that of the nodes brought by u.

The evaluation of the obtained CQL program is in PTIME in the size of the data. We show that the combined complexity of the problem is NP-HARD. This is not surprising since constraints on time induce some form of join. We prove that query satisfaction, for time-based documents and queries, is NP-HARD by reduction of 3SAT. Since query satisfaction can be reduced to document satisfiability, this shows that document satisfiability is NP-HARD.

Let φ be 3SAT formula. For each variable x of φ, let $a_x, x, \bar{x}$ be three distinct new labels. The document has root labeled r with a child labeled a_x for each variable x occurring in φ. The a_x child of r has four children, two labelled $x, \bar{x}$ with time annotation 1, and two also labelled $x, \bar{x}$ with time annotation 0. The query q also has root labeled r with a child labeled a_x for each variable x occurring in φ. The a_x-child of r in q, has two children labeled x and $\bar{x}$. With the abuse of notation of confusing a node labeled l with its label (since no two nodes in q have the same label), the time constraints are:

- for each variable x, $x + \bar{x} = 1$
- for each clause $l_1 \vee l_2 \vee l_3$, $l_1 + l_2 + l_3 \geq 1$

Clearly, the document satisfies the query iff the formula is satisfiable. □

We next turn to monotone extensions of the query language.

Disjunction.

One could consider queries with disjunction. For instance, one could introduce some particular nodes that specify that *one of the subtree-patterns has to match*. Since datalog captures disjunction, the results from the previous section can easily be extended to query languages with disjunction.

Nonmonotonicity.

To conclude this section, we consider lifting the monotonicity restriction.

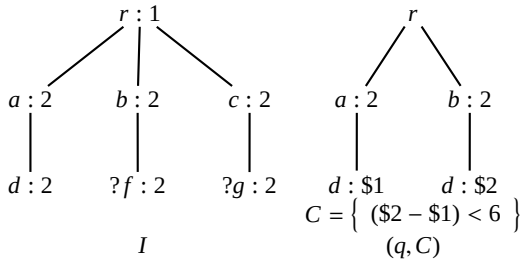
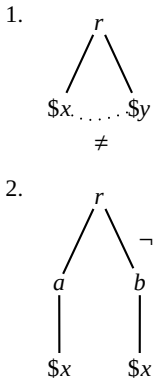


Figure 6: Time-based instance and query

In general, nonmonotone queries (negative patterns) and nonmonotone documents (deletions) complicate the problem. A positive formula as previously defined may become false as a consequence of a deletion; then turn back to true because of some insertion; etc. Also, a query with negation may flip-flop between true and false as a consequence of inserts only. This suggests moving to richer fragments of temporal logic and is left for future research. To illustrate the increased complexity, we consider queries with negation. We consider that some edges are labeled with ¬ with the meaning that the pattern should not exist and introduce inequalities between variables as join conditions with the obvious meaning. For instance, we could consider the query:



i.e., 1. the root has two children with different labels; and 2. the label of a child of an a-child of the root is not the label of any child of its b-children. The quantification of variables is as follows. Variables not occurring in the output are existentially quantified in the least ancestor of the nodes where they occur. (We also impose that a variable occuring in a negative pattern should be positively bound externally to the pattern.) In general, we have:

THEOREM 8. *Document satisfiability is undecidable for queries with negation even for documents with bounded depth.*

PROOF. The proof is by reduction of the implication problem of functional and inclusion dependencies [20]. It is given in appendix. □

It is interesting to obtain restrictions that make document satisfiability decidable even in presence of negation. Such restrictions are considered in [6].

With respect to deletion, consider functions that return streams of *delete sibling* messages. We can define a dual notion to satisfiability for a document, namely *formula nonpersistence*, that specifies that a formula may become unsatisfied in some future of the document. For such formulas only deletions matter. Techniques

similar to those used for satisfiability may be applied. In particular, formula nonpersistence may be evaluated by nrec-datalog with negation programs. Details omitted.

5. INCREMENTAL MAINTENANCE

We are interested in the management of queries over active documents. In particular, we want to maintain incrementally the results of queries. We return to monotone queries (that we defined simply as queries) and inserts only. To simplify, we consider here only Boolean queries. But this can be generalized easily to arbitrary ones. Given I and q, one of the following three cases occurs:

(1: true) I satisfies q and it will always do, $I \models q$.

($\frac{1}{2}$: false but possible) I does not satisfy q but it may in the future, i.e., $I \not\models q \wedge I \models \diamond q$.

(0: forever false) I will never satisfy q, $I \not\models \diamond q$.

We maintain incrementally the results of both q and $\diamond q$. For each incomplete tuple, we have a pair of Boolean values that may be interpreted in 3-valued logic as: true, false but possible, forever false. For this maintenance, we use datalog. We briefly mention in this section how rather standard known datalog techniques can be used to perform and optimize incremental maintenance. We will introduce in the next section other optimization techniques for view maintenance. The two known techniques we use are:

Differential [10] is a technique for efficient maintenance of datalog-defined views that is based on differentiating the datalog program to avoid deriving several times the same data.

MagicSet [8, 26] is a technique for efficient evaluation of datalog queries that is based on rewriting the program into another program that derives only relevant data.

Suppose that we want to maintain a view defined by a query over some active document. Each time the document changes, we have to reevaluate the query. We can take advantage of datalog technology as follows. Consider a query q. One can first use a datalog program that computes the satisfiability q. One can then optimize this program using the MagicSet technique. One can finally optimize its maintenance using the Differential technique. Although this is much better than simply reevaluating the query at each document update, this straightforward idea presents a serious limitation that provides already some motivation for the notion of satisfiability of a document. This limitation and a solution based on satisfiability are discussed next.

Consider the query q and instance I in Figure 7. The data consists of a large collection of nodes labeled b'' (brought by $?g$). Observe that until $?f$ produces a node labeled y (for yes), they do not produce any answer. If we evaluate the query q using datalog and MagicSet, no tuple is produced until a node labeled y is received. Indeed, the b' subtrees are not even tested until a node labeled y is received. This would work fine in absence of the function $?f$. But with the presence of $?f$, we know that this function may return a node labeled y, so it seems more appropriate to be optimistic and start testing the b' subtrees. This is indeed what happens if we consider satisfiability. Since the y-subquery is satisfiable, we process the b' subtree. When a node labeled y arrives, there is almost no computation needed to derive that 1,2,... are answers. Such a more aggressive (and optimistic) computation presents the disadvantage that we may derive facts unnecessarily. It presents the advantages that we balance the query load more evenly and that we can react much faster to the arrival of new data. We will see in the next section that the technique also enables further optimization.

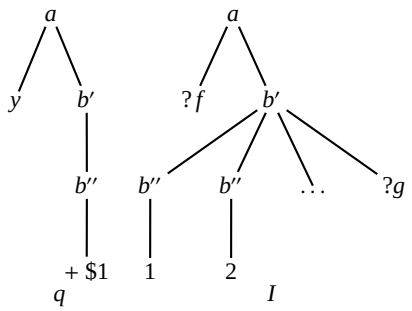


Figure 7: Beyond MagicSet

More precisely, the technique consists in maintaining the query computing the satisfied tuples and that computing the satisfiable incomplete tuples, using both Differential and MagicSet. As a by product, we also have the satisfiable answers. Note that as in the Differential technique, we have to eliminate duplicates if desired by the application.

An example of 3-valued datalog program is shown in Algorithm 1. This is the program *before* applying Differential since Differential works essentially the same for the MagicSet setting and in our setting. Our 3-valued datalog program derives essentially the same facts as MagicSet if MagicSet were applied to a 2-valued datalog program computing both satisfaction and satisfiability. It turns out that the use of a 3-valued logic makes the derivation more compact, since we sometimes "share" computations between satisfaction and satisfiability.

Algorithm 1: δ_q for the simple query $q = a[b, c]$ of Figure 3

begin

$q() \leftarrow a(n)$
$\widehat{a}(n) \leftarrow root(n), label(a, n)$
$\widehat{b}(n) \leftarrow \widehat{a}(n'), child(n', n), label(b, n)$
$b(n) \leftarrow \widehat{a}(n'), child(n', n), function(n), \frac{1}{2}$
$c(n) \leftarrow \widehat{a}(n'), child(n', n), label(c, n), b(m) \geq \frac{1}{2}, child(n', m)$
$c(n) \leftarrow \widehat{a}(n'), child(n', n), function(n),$
$\quad b(m) \geq \frac{1}{2}, child(n', m), \frac{1}{2}$
$a(n) \leftarrow b(n'), c(m), \widehat{a}(n), child(n, n'), child(n, m)$

end

We conclude this section with two remarks related to limitations of the incremental maintenance of satisfiability in datalog. The first one is related to the notion of reduced document.

Remark: Consider a document $r[a[b]][a[?f]]$. Observe that the root has two children. Now suppose that $?f$ returns a b-node. Then the reduced new instance is $r[a[b, ?f]]$ where the root node has a single child. Since we cannot compute tree homomorphism in datalog, our datalog programs are unable to apply such reductions. So, the documents that they will simulate may become unreduced as the result of some insertions. This does not prevent the datalog program from computing the correct answers. □

The second (and last) remark is on the nonmonotonicity of satisfiability.

Remark: Observe the monotonicity of the 3-valued satisfiability for the queries and active documents defined in Section 2. A formula that is 0 or 1 will remain so forever. And a formula that is $\frac{1}{2}$, may change to 1. Now consider extensions we introduced. With features that are not monotone for formula satisfaction (negation in queries and deletions), the monotonicity of satisfiability is

clearly lost. Now consider other features that are monotone for formula satisfaction. For non-stream functions, for functions with end-of-stream, and in the presence of document or function typing, the monotonicity of satisfiability is lost. For instance, some end-of-stream may turn a formula from $\frac{1}{2}$ to 0. We observed a similar situation for time-based queries. This nonmonotonicity may be viewed as a fundamental explanation for the increase in complexity. It clearly complicates view maintenance since satisfiability statements that have been derived may be invalidated by some incoming data. We will see how this is handled in our implementation. □

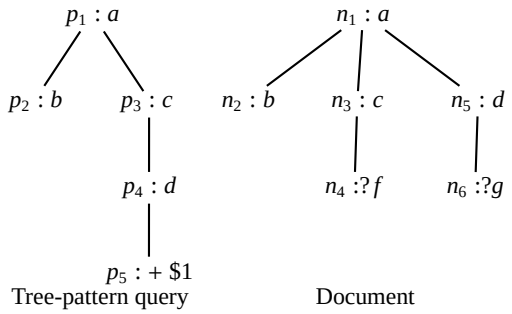


Figure 8: Filter pushing example

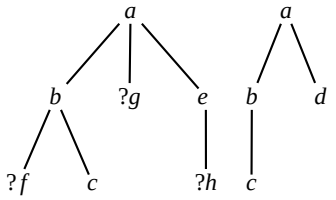


Figure 9: A document d and a query q

6. FURTHER OPTIMIZATION

We present in this section a new technique to further optimize the incremental maintenance of active documents. The technique consists primarily in filtering the stream of data produced by the functions occurring in the active document to reduce the amount of data entering the document. This technique, in the spirit of "pushing selections" in relational database optimization, presents several advantages. First, it may save in communications when the filtering can be pushed to a distant machine. Also, it may save processing time by using efficient stream filtering techniques such as YFilter [12]. Finally, when no relevant selection is found for a function, then this function becomes useless (for this particular task) and further saving can be obtained by ignoring it. We sketch the technique in this section.

Let n be a node of the document labeled by some function $?f$ and q a node of the query (denoting the subquery rooted at that node). We say that q is *relevant* for n if some data matching q returned by $?f$ would bring some change to the result (with respect to satisfiability). Note that for a global Boolean query q_0, if $I \models q_0$ then there is no such relevant pair (q, n) and similarly if $I \not\models \diamond q_0$. Relevance really means a chance to change the result.

Then we say that q is *relevant* for $?f$ if it is relevant for some n labeled $?f$. Now consider some function $?f_i$ used in the document and some q_j relevant for $?f_i$. We can filter the stream of data produced by $?f_i$ with q_j using a filter that essentially computes the tree-pattern query q_j on the input[4]. The filter produces a stream of tuples that we feed directly into the datalog program. We do that for each i, j.

Observe that all the filters for a function $?f_i$ can be performed simultaneously by a single filter. Observe also that we now have several filters (one for each $?f_i$) feeding data directly to the datalog program. One can verify that the resulting system produces the same output as the original datalog program.

Consider the query and the document in Figure 8, where the p_i and n_j represent node identifiers of the two trees. Observe that before we can derive some result, the pattern rooted at p_4 has to be matched to data returned by $?f$. This leads to installing a p_4 filter on $?f$. Suppose that this filter produces tuples in a relation $filter^{p_4}_{?f}$. This relation is unary because we only need to keep one value per matching, namely for \$1. Suppose also that our datalog program computes a monadic relation $relevant^{p_4}_{?f}$ with the obvious meaning. Then the interface between the filter and the datalog program consists in the single rule:

$$\widehat{p_4}(x, \$1) \leftarrow relevant^{p_4}_{?f}(x), filter^{p_4}_{?f}(\$1)$$

We will have one such rule for each $(p_i, ?f_j)$ pair.

Now suppose that no q_j was found relevant for a function $?f$. Then one can rightly conclude that this function is not useful for the maintenance of this view. In particular, if we have no other use of this function, we can close this function, which potentially leads to communication, processing and financial savings (e.g., subscription cost).

The previous discussion suggests a definition of usefulness that we propose next. To simplify, we do it for Boolean queries but it can easily be generalized to arbitrary queries. Given a sequence ω of updates and a function $?f$, let $\omega_{no\text{-}f}$ denote the sequence obtained from ω by removing all $?f$-insertions. Now, we have:

DEFINITION 5. *Let q be a Boolean query and I an instance such that $I \models \diamond q$ and $I \not\models q$. A function $?f$ is said to be* useless *for q and I iff for each update sequence ω, $\omega(I) \models q$ iff $\omega_{no\text{-}f}(I) \models q$.*

Intuitively, a function $?f$ is useless for I, q if ignoring the outputs of $?f$ in I does not change the result of q.

The example in Figure 9 illustrates this notion. Note that $?h$ is useless because its e parent does not match, neither b nor d. Also, $?f$ is useless because some sibling already provides the c. On the other hand, $?g$ is clearly useful since it can bring the node labeled d.

One could believe that one can check in datalog whether a function is useful or not. This turns out to be false. The example in Figure 10 illustrates a subtlety of the problem. Consider the sequence $\omega = (?f, e[c]), (?g, c)$ that yields a document satisfying the query. A superficial analysis would lead to believe that the call to $?f$ was useful because it brought data that matched part of the query. However, observe that the update $(?g, c)$ alone is enough to yield a document satisfying the query. So the update $(?f, e[c])$ is not really needed in that particular sequence. One can show more generally that $?f$ is useless for the document and the query in Figure 10.

Indeed, we have:

THEOREM 9. *The problem of deciding, given a document and a no-join Boolean query, whether a function is useful, is* NP-COMPLETE *in the size of the query and the document.*

[4]In the implementation, we first treat the pattern without join variable. We then postprocess to take them into account.

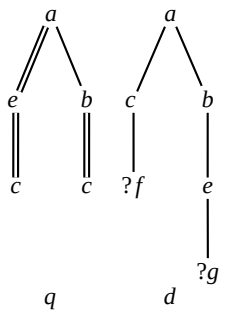


Figure 10: Example of hard usefulness

The problem of deciding the usefulness of a function,given a document and a no-join Boolean query, is PTIME *in the size of the document.*

The proof is a variation of previous proofs and is given in appendix. For the PTIME bound, it again consists in computing scenarios.

Remark: The idea of removing streams that cannot bring useful information can be also used for data. When the data in a subtree is no more useful (after all tuples that could be derived with it have been derived), it is not necessary to keep it. The subtree can then be garbaged collected using techniques similar to those used in this section for useless functions. □

7. IMPLEMENTATION

The present work is motivated by the development of a P2P monitoring system based on flows of (A)XML data and active documents [4].

We have developed a prototype that maintains views of active documents. This prototype uses the incremental technique sketched in Section 5 based on satisfiability, Differential and MagicSet. As in Section 6, it also pushes filters to streams and eliminates useless functions, with useless computed simply in a sufficient manner. (Useless functions may still remain.) The implementation is based on the *eXist* [13] native XML database system supporting XQuery. The active documents are persistent as well as some relations corresponding to intentional predicates of the datalog program. One of the relations, namely *result*, holds the current value of the view.

Although datalog has been intensively studied, datalog implementations are not easily available. Furthermore, a first toy implementation based on datalog we developed showed to be rather inefficient. So, we centered our implementation around XQuery. We used datalog in the presentation of the paper for clarity and simplicity. We use XQuery at different levels. First, in a non-incremental way, when a new document and a query q are checked in the system: (i) we compute with XQuery the satisfiability of incomplete tuples as answers to q (i.e., evaluate q and $\diamond q$); and (ii) we compute the useless functions (in a sufficient way) and eliminate them. Then, in an incremental manner, when some new data arrives from some function $?f_i$: (iii) we run the filters on this data; and (iv) when some data pass the filter for some $?f_i$, say $\Delta?f_i$, we use an XQuery statement S_i to incrementally propagate the changes. This statement (that depends on the document and the persistent relations) appends tuples to the relations, and may derive some new tuples in *result*.

Recall the remark on the nonmonotonicity of satisfiability. The information on useful data also introduces some nonmonotonicity. Our system maintains incrementally information concerning satisfiability and usefulness. To take the nonmonotonicity of satisfiability (and usefulness) into account, the system periodically reevaluates (refresh) our current knowledge for both. (Thus, we regularly reevaluate (i-ii) to save effort in (iii-iv).) Clearly, one could also consider more complex view maintenance in the style of the datalog view maintenance in presence of deletion, that would maintain this information incrementally. This is left for future research.

Finally, the current implementation assumes that the input streams are passive. More work is planned in this direction.

8. CONCLUSION AND RELATED WORKS

Our work relies on previous works around incremental view maintenance [10, 19, 18]. We used datalog to benefit from known techniques, Magic Set [8] (similarly, QSQ [26]) incremental evaluation (see [3]) and constraint databases [21]. Connections between tree-pattern queries and monadic datalog have been studied in [16]. Tree-pattern queries over trees are closely related to XPATH expressions but there are differences in expressive power; see e.g., [22]. The problem of the incremental view maintenance for graph-shaped semistructured data is studied in [5]. Some recent works have considered the incremental maintenance of XPATH views over trees [25, 24].

Our complexity analysis for query evaluation on active documents has been influenced by works of [16, 17, 22]. In particular, the complexity for datalog evaluation on trees is studied in [16] whereas [17] considers the complexity of query evaluation for portions of XPATH; and [22] does the complexity of boolean tree-pattern query evaluation on trees and tree-pattern query containment.

Query evaluation for active documents is studied in [1]. The context is essentially different since their functions are non-stream and they do not consider incremental maintenance, but only query evaluation. Other works about active documents may be found at [7]. The work presented here was mostly about positive AXML. It is clearly interesting to study nonpositive extensions. Some results in that direction are presented in [6].

The system we mentioned is running. It already supports queries over active documents as described in Section 2. The optimizations of Sections 5 and 6 are already used by the system. We are currently working on extending it with (i) streams with active results, (ii) terminating functions, and (iii) time-based queries and documents. We are also investigating other optimization techniques of a more automata-theoretic flavor.

We mentioned a number of situations where nonmonotonicity gets in the way of incremental maintenance. Indeed, in the current prototype, we sometimes need to "refresh" documents, e.g., to recompute the usefulness of functions. It would be interesting to investigate the use of datalog with negation, e.g. with well-founded semantics [15], in this setting.

9. REFERENCES

[1] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy query evaluation for Active XML. In *SIGMOD Conference*, pages 227–238, 2004.

[2] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The Active XML project: an overview. *VLDB J.*, accepted for publication, 2008.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Serge Abiteboul and Bogdan Marinoiu. Distributed Monitoring of Peer to Peer Systems. In *Workshop On Web Information And Data Management*, pages 41–48, 2007.

[5] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[6] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active documents, 2007.

[7] Active XML, http://activexml.net.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. pages 269–284, 1987.

[9] Michael Benedikt, Wenfei Fan, and Floris Geerts. Xpath satisfiability in the presence of dtds. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[10] Stefano Ceri and Jennifer Widom. Deriving incremental production rules for deductive data. *Inf. Syst.*, 19(6):467–490, 1994.

[11] 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.

[12] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of XML documents. In *ICDE*, pages 341–, 2002.

[13] eXist, http://exist.sourceforge.net/.

[14] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[15] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.

[16] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *LICS*, pages 189–202, 2002.

[17] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[18] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[19] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

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

[21] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *PODS*, pages 299–313, 1990.

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

[23] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[24] Makoto Onizuka, Fong Yee Chan, Ryusuke Michigami, and Takashi Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW*, pages 671–681, 2005.

[25] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal, and K. Selçuk Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, pages 443–454, 2005.

[26] L. Vielle. Recursive query processing: the power of logic. *Theor. Comput. Sci.*, 69(1):1–53, 1989.

APPENDIX

Notation.

A *data tree* is a document or a tree that consists of a single node labeled by a function. Let q, t be a query, a data tree, and p, n some nodes, respectively in q, t. Then:

- $desc_q(p)$ is the set of descendants of p.
- $\lfloor p \rfloor_q$ and $\lfloor n \rfloor_t$ are the subtrees rooted at p and n, respectively.
- $//q$ is the query with a root that (i) is labeled $*$; (ii) has q as a single subtree ; and (iii) the edge from the root is in $E_{//}$.

Finally, $[p]_q$ is the query defined as follows:

- if the edge leading to p is in $E_/$, then $[p]_q = \lfloor p \rfloor_q$.
- if the edge leading to p is in $E_{//}$, then $[p]_q = \lfloor p \rfloor_q \vee //(\lfloor p \rfloor_q)$.

Figure 11 shows a query q and the derived queries : $\lfloor c \rfloor_q$, $//\lfloor c \rfloor_q$ and $[c]_q$. When q, t are understood, we use $desc(p)$, $\lfloor p \rfloor$ and $\lfloor n \rfloor$, $[p]$.

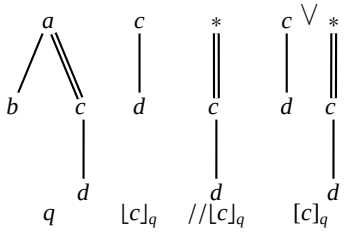


Figure 11: Examples of queries

Proof of Theorem 1.

The following lemma is at the core of the inductive construction of datalog programs that check document satisfiability.

LEMMA 1. *Let t be a data tree and q a no-join Boolean query. Then $t \models \diamond q$ iff one of the two following conditions is true:*

- $\lambda_t(root(t)) \in \mathcal{F}$
- $\lambda_q(root(q)) \in \mathcal{V}$ *or* $\lambda_q(root(q)) = \lambda_t(root(r))$, *and:*
  1. *for each child p of $root(q)$ such that $(root(q), p) \in E_/$, there exists a child n of $root(t)$ such that the following is true for n's associated subtree: $\lfloor n \rfloor \models \diamond \lfloor p \rfloor$*
  2. *for each child p of $root(q)$ such that $(root(q), p) \in E_{//}$ there is a descendant n of $root(t)$ such that $\lfloor n \rfloor \models \diamond \lfloor p \rfloor$*

Algorithm 2, given further, associates to each query q, an nrec-datalog program δ_q such that $I \models \diamond q$ iff $\delta_q(I)$ is not empty. The algorithm uses extensional relations *root, child, descendant, label, function* with the obvious meaning. Each node p of the query q is associated to an intentional relation with the same name in δ_q. From Lemma 1, it follows that, for each relation p, $p(n)$ iff $\lfloor n \rfloor \models \diamond \lfloor p \rfloor$. □

Proof of Theorem 2.

We use Algorithm 3, an extension of Algorithm 2, to construct datalog programs that compute document satisfiability in PTIME in the size of the document. We briefly explain some aspects of the datalog programs and sketch a proof of correctness.

In the program, we denote by $Activevar(p)$ the set of variables that appear in $\lfloor p \rfloor$ and that are needed in the output or that need to be carried on for checking a join constraint. We denote by $Join(p)$ the nodes such that some join conditions involving them must be checked at p. Obviously the two sets can be computed in a preprocessing phase, ignoring the data.

The datalog program computes with incomplete tuples. For the variables in incomplete tuples, we use the variables in the query. We use two new relations Var and $Label$, such that $Var(x)$ means that x is a variable (in q) and $Label(x)$ means that x is a label (in q or d).

To simplify, we will assume without loss of generality that the same variable does not occur more than once in output nodes and that output nodes are all labeled by variables. An output of the datalog program may then be seen as a tuple over the variables occurring in the query. The entry for variable $\$i$ of a tuple is either $\$i$ or a label. An intermediary result (corresponding to a subtree of the query tree) is such a tuple. (In Algorithm 3, the notation $u^{\$i}$ denotes the component of the tuple u that corresponds to the variable $\$i$.) We have to join tuples. Note that by construction of the algorithm, we may have to unify a variable $\$i$ with constants, but never $\$i$ with $\$j$ for $i \neq j$. The unification of 2 terms is computed by the following rules:

$$\begin{array}{lcl} Unif_2(x,x,x) & \leftarrow & Label(x) \\ Unif_2(x,x,x) & \leftarrow & Var(x) \\ Unif_2(x,x,x') & \leftarrow & Label(x), Var(x') \\ Unif_2(x,x',x) & \leftarrow & Label(x), Var(x') \end{array}$$

When joining i branches ($i \leq |q|$), one may also have to perform joins of up to i terms. These are computed by the following rules:

$$\begin{array}{lcl} Unif_i(x,x,x_2,\ldots,x_i) & \leftarrow & Unif_{i-1}(x,x_2,\ldots x_i) \\ Unif_i(x,x,x_2,\ldots,x_i) & \leftarrow & Label(x), Var(x'), \\ & & Unif_{i-1}(x',x_2,\ldots x_i) \\ Unif_i(x,x_1,\ldots,x_i) & \leftarrow & Label(x), Var(x_1), \\ & & Unif_{i-1}(x,x_2,\ldots,x_i) \end{array}$$

The next lemma states the soundness of the datalog program.

LEMMA 2. *Let q, δ_q, I, n and u be a query, the datalog program associated to q, an instance, a node of the instance and an incomplete tuple. Let p be a node of q. If $(n, u) \in p$ in $\delta_q(I)$, then $\lfloor n \rfloor \models \diamond \lfloor p \rfloor(u)$.*

PROOF. We prove it by induction.

Let p be a leaf of q. If p is labeled by a variable $\$i$, then $Activar(p) = \{\$i\}$ if $p \in \pi$ or if $\$i$ appears at least twice in the query (one needs to carry on this value for checking join constraints). So $(n, \$i) \in p$ iff n is a function node, and $(n, a) \in p$ iff n is a data node labeled by a.

Let p and p_{i_x} its children such that $(p, p_{i_x}) \in E_/$ (and similarly for $E_{//}$). Let u be a tuple and n be a data node such that $(n, u) \in p$. It is obvious that if n is a function node, then all queries are satisfiable. In particular, u verifies that $\lfloor n \rfloor \models \diamond \lfloor p \rfloor(u)$.

If n is a data node, there is $(n_x, u_x) \in p_{i_x}$. By induction, for all instantiation f_x of $var(u_x)$, there is a sequence ω_x such that $\omega_x(\lfloor n_x \rfloor) \models q(f_x(u))$. It is easy to see that because of unification, for all instantiation of u, there is a possible derived instantiation for each u_x.

So, for each instantiation of u, there is a sequence which is the concatenation of all sequences ω_x and ω_y.

Therefore $\lfloor n \rfloor \models \diamond \lfloor p \rfloor(u)$. □

Finally, the next lemma states the completeness of the datalog program.

LEMMA 3. *Let q, δ_q, I, u be a query, the datalog program associated to q, an instance and an incomplete tuple. Let p and n be a*

Algorithm 2: Satisfiability for no-join Boolean queries

Data: a query q

Result: a datalog program computing satisfiability δ_q

begin

  for $p = root(q)$ **do**

    δ +:= $q() \leftarrow p(n), root(n)$

  foreach $p \in nodes(q)$ **do**

    if p *labeled by* $b \in \mathcal{L}$ **then**

      δ +:= $p(n) \leftarrow label(b, n), p_{i_1}(n_1), \ldots, p_{j_1}(n'_1), \ldots, child(n, n_1), \ldots, descendant(n, n'_1), \ldots$
      where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$

    else

      δ +:= $p(n) \leftarrow label(x, n), p_{i_1}(n_1), \ldots, p_{j_1}(n'_1), \ldots, child(n, n_1), \ldots, descendant(n, n'_1), \ldots$
      where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$

  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**

    δ +:= $p(n) \leftarrow function(n)$

end

Algorithm 3: Satisfiability for queries

Data: a tree-pattern query q

Result: the datalog program δ_q

begin

  for $p = root(q)$ **do**

    δ +:= $q() \leftarrow p(n), root(n)$

  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**

    if $Join(p) \neq \phi$ **then**

      if p *labeled by* $b \in \mathcal{L}$ **then**

        δ +:= $p(n, u^{\$z}, \ldots) \leftarrow label(b, n), p_{i_1}(n_1, u_{i_1}), \ldots, p_{j_1}(n'_1, u_{j_1}), \ldots,$
        $child(n, n_1), descendant(n, n'_1), \ldots, Unif_{l_{\lambda(p)}}(u^{\lambda(p)}, b, u_\alpha^{\lambda(p)}, \ldots) Unif_{l_{\$k}}(u^{\$k}, u_\alpha^{\$k}, \ldots), \ldots$
        where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
        $\$k \in \lambda(Join(p))$, $p_\alpha \in Join(p) \cap \lambda^{-1}(\$k)$, $l_{\$k} = |Join(p) \cap \lambda^{-1}(\$k)|$,
        $\$z \in Activevar(p)$,
        $\forall \$w \in Activevar(p) - Join(p), \beta \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\} : u^{\$w} = u_\beta^{\$w}$

      else

        δ +:= $p(n, u^{\$z}, \ldots) \leftarrow label(x, n), p_{i_1}(n_1, u_{i_1}), \ldots, p_{j_1}(n'_1, u_{j_1}), \ldots,$
        $child(n, n_1), descendant(n, n'_1), \ldots, Unif_{l_{\$k}}(u^{\lambda(p)}, x, u_\alpha^{\lambda(p)}, \ldots,), Unif_{l_{\$k}}(u^{\$k}, u_\alpha^{\$k}, \ldots), \ldots$
        where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
        $\$k \in \lambda(Join(p))$, $p_\alpha \in Join(p) \cap \lambda^{-1}(\$k)$, $l_{\$k} = |Join(p) \cap \lambda^{-1}(\$k)|$,
        $\$z \in Activevar(p)$
        $\forall \$w \in Activevar(p) - Join(p), \beta \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\} : u^{\$w} = u_\beta^{\$w}$

    else

      if p *labeled by* $b \in \mathcal{L}$ **then**

        δ +:= $p(n, u^{\$z}, \ldots) \leftarrow label(b, n), p_{i_1}(n_1, u_{i_1}), \ldots, p_{j_1}(n'_1, u_{j_1}), \ldots,$
        $child(n, n_1), descendant(n, n'_1), \ldots$
        where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
        $\$z \in Activevar(p)$,
        $\forall \$w \in Activevar(p) - \{\lambda(p)\}, \beta \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\} : u^{\$w} = u_\beta^{\$w}$
        $\lambda(p) \in Activevar(p) \implies u^{\lambda(p)} = b$

      else

        δ +:= $p(n, u^{\$z}, \ldots) \leftarrow label(x, n), p_{i_1}(n_1, u_{i_1}), \ldots, p_{j_1}(n'_1, u_{j_1}), \ldots,$
        $child(n, n_1), descendant(n, n'_1), \ldots$
        where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$,
        $\$z \in Activevar(p)$,
        $\forall \$w \in Activevar(p) - \{\lambda(p)\}, \beta \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\} : u^{\$w} = u_\beta^{\$w}$
        $\lambda(p) \in Activevar(p) \implies u^{\lambda(p)} = x$

  foreach $p \in nodes(q) \setminus \{root(q)\}$ **do**

    δ +:= $p(n, u) \leftarrow function(n)$
    where $\forall \$i \in Activevar(p), u^{\$i} = \$i$

end

node of q and a node of I. If $u \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$ then there exists a tuple u' produced by $\delta_q(I)$ such that $u' \Rightarrow u$ and $(n, u') \in p$.

PROOF. We prove by induction on q.

Let p and u be a leaf of q and a tuple such that $u \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$.

If $u = (\$i)$ then n is a function node. Otherwise, if $u = (a)$ then n is labeled by a. If p is labeled by a variable $\$i$, then $Activar(p) = \{\$i\}$ if $p \in \pi$ or $\$i$ appears at least twice in the query. So $(n, \$i) \in p$ iff n is a function node, and $(n, a) \in p$ iff n is a data node labeled by a.

Let p and p_{i_x} its children such that $(p, p_{i_x}) \in E_/$. Let u and n be a tuple and a data node such that $u \in \diamond\lfloor p \rfloor(\lfloor n \rfloor)$.

If n is a function node, it is obvious that there is u_1 such that $u_1^{\$i} = \$i$. So, $u_1 \Rightarrow u$.

If n is not a function node, there is an incomplete tuple u_x and child n_x such that $u_x \in \diamond\lfloor p_{i_x} \rfloor(\lfloor n_x \rfloor)$. The same is true for $(n_y, u_y) \in p_{i_y}$.

Let u be the unification of all u_x and u_y. By induction, there is u'_x such that $(n_x, u'_x) \in p_{i_x}$ and $u'_x \Rightarrow u_x$ then u' the unified tuple of u'_x. The unification operation is obviously monotone. So $u' \Rightarrow u$. □

Proof of Theorem 3.

Let q, I, u be a query, an instance, and an incomplete tuple. By definition, $I \models \diamond q(u)$ iff for all instantiations θ of u, $I \models \diamond q(\theta(u))$. We show that it suffices to check one particular instantiation.

LEMMA 4. *Let q, I, u be a query, an instance and an incomplete tuple. Let θ be a particular instantiation of u that associates to each $\$i$, a distinct, fresh label i.e. a label not appearing in I or q. Then $I \models \diamond q(u)$ iff $I \models \diamond q(\theta(u))$.*

PROOF. ($\Rightarrow$) If $I \models \diamond q(u)$, for all instantiation of u, θ', $I \models \diamond q(\theta'(u))$. Thus, in particular, $I \models \diamond q(\theta(u))$.

($\Leftarrow$) Now suppose that $I \models \diamond q(\theta(u))$ for this particular θ. Let ω be an update sequence so that $\omega(I) \models q(\theta(u))$. Let θ' be an arbitrary instantiation. By construction of θ, there exists a function f such that for all $\$i$, $\theta'(\$i) = f(\theta(\$i))$. Let ω' be the update sequence obtained from ω by replacing each constant $\theta(\$i)$ by $\theta'(\$i) = f(\theta(\$i))$. Clearly, $\omega'(I) \models q(\theta'(u))$, so $I \models \diamond q(\theta'(u))$. Since θ' is arbitrary, $I \models \diamond q(u)$. □

Scenarios.

The computation of scenarios is done by a datalog program named *ScenariosBuilder* , that is provided by the Algorithm 4.

The datalog program, namely *ScenariosBuilder*, computes *ScenariosSet*, a particular set of scenarios. This is done as follows. After computing satisfiability, we record which functions could bring data that match which subqueries. The size of a relation corresponding to a node p in the query (relation noted $\tilde{p}$) is the number of nodes in the subquery rooted at p. The function name is stored in a relation $label\_function(x, n)$, where n is the function node and x is the function name. The details are omitted. By construction, each possible scenario will be in *ScenariosSet*, i.e., this set provides a necessary condition for usefulness. However, some of these scenarios may involve calls to functions that are not useful, so the condition it provides is not sufficient for usefulness. This is for instance the case for the example in Figure 10.

Proof of Theorem 6.

Hard: Let $\varphi = \wedge_{i\in[1..n]} C_i$ be a 3SAT formula. We can assume without loss of generality that no clause contains a variable x_j and its negation $\bar{x_j}$ and that each variable and its negation appear in the formula.

Algorithm 4: ScenariosBuilder for no-join Boolean queries

Data: a query q, relations p of *Satisfiability*(q)
Result: the datalog program δ = *ScenariosBuilder*(q)
begin
  for $p = root(q)$ *of label* a **do**
    Top-down evaluation
    δ +:= $ScenariosSet(u) \leftarrow \tilde{p}(n, u)$
    δ +:= $relevant_p(n) \leftarrow root(n), (p(n) = \frac{1}{2})$
    Bottom-up evaluation
    δ +:= $\tilde{p}(n, \perp, u_1, \cdots, u'_1, \cdots) \leftarrow relevant_p(n)$,
      $\tilde{p}_{i_1}(n_1, u_1), \cdots, \tilde{p}_{j_1}(n'_1, u'_1), \cdots, child(n, n_1), \cdots,$
      $descendant(n, n'_1), \cdots$
        where $(p, p_{i_x}) \in E_/, (p, p_{j_y}) \in E_{//}$
  foreach $p \in nodes(q) \setminus \{root(q)\}$ *of label* b **do**
    Top-down evaluation
    if *the arc between them is direct child* **then**
      δ +:= $relevant_p(n) \leftarrow relevant_{p'}(n'), child(n', n),$
        $(p(n) = \frac{1}{2}), \nexists n''((p(n'') = 1), child(n', n''))$
    else
      δ +:= $relevant_p(n) \leftarrow relevant_{p'}(n'),$
        $descendant(n', n), (p(n) = \frac{1}{2}),$
        $\nexists n''((p(n'') = 1), descendant(n', n''))$
    Bottom-up evaluation
    δ +:= $\tilde{p}(n, \perp, u_1, \cdots, u'_1, \cdots) \leftarrow relevant_p(n), \tilde{p}_{i_1}(n_1, u_1),$
      $\cdots, \tilde{p}_{j_1}(n'_1, u'_1), \cdots, child(n, n_1), \cdots, descendant(n, n'_1), \cdots$
        where $(p, p_{i_x}) \in E_/, (p, p_{j_y}) \in E_{//}$
    δ +:=$\tilde{p}(n, x, \perp, \cdots, \perp) \leftarrow relevant_p(n), function(n),$
      $label\_function(x, n)$
    δ +:=$\tilde{p}(n, \perp, \perp, \cdots, \perp) \leftarrow (p(n) = 1)$
end

From φ, we construct an instance of the document-satisfiability problem, i.e., a document I_φ and a query q_φ, as follows. For each variable x, let x be a distinct label. For each C_i, let c_i be also a distinct new label. The document uses some functions $?h_x$ and $?h_{\bar{x}}$ for each variable x. The active document I_φ is as follows: the root, labeled r, has one subtree t_x for each variable x. The subtree t_x has a root labeled x and two subtrees, $\tau_x, \tau_{\bar{x}}$ defined as follows:

- The tree τ_x has a root labeled x, one of its children is labeled by the function $?h_x$ and its other children are labeled by c_i where the literal x appears in C_i.
- The tree $\tau_{\bar{x}}$ has a root labeled $\bar{x}$, one of its children is labeled by the function $?h_{\bar{x}}$ and its other children are labeled by c_i where the literal $\bar{x}$ appears in C_i.

The query q_φ has its root labeled r and has one subtree for each variable denoted by q_x and one subtree for each clause C_i denoted by q_{C_i} built as follows:

- q_x is $x[*[0]][*[1]]$
- q_{C_i} is $*[*[c_i][1]]$

The instance of document $I(d)$ and the query q are shown in Figure 12 for

$$\varphi = \underbrace{(x_1 \vee x_2 \vee \bar{x_3})}_{C_1} \wedge \underbrace{(\bar{x_1} \vee \bar{x_2} \vee x_3)}_{C_2} \wedge \underbrace{(x_1 \vee \bar{x_2} \vee x_3)}_{C_3}$$

One can show that q_φ is satisfiable for I_φ iff φ is satisfiable. The proof is as follows:

$\Rightarrow$ Let ω be an update sequence so that $\omega(I)$ satisfies the query. Clearly, each function must have received some data. For each x, if $?h_x$ received a tree with root labeled 1, let $v(x)$ be true. Otherwise, let it be false. It is easy to see that v satisfies φ.

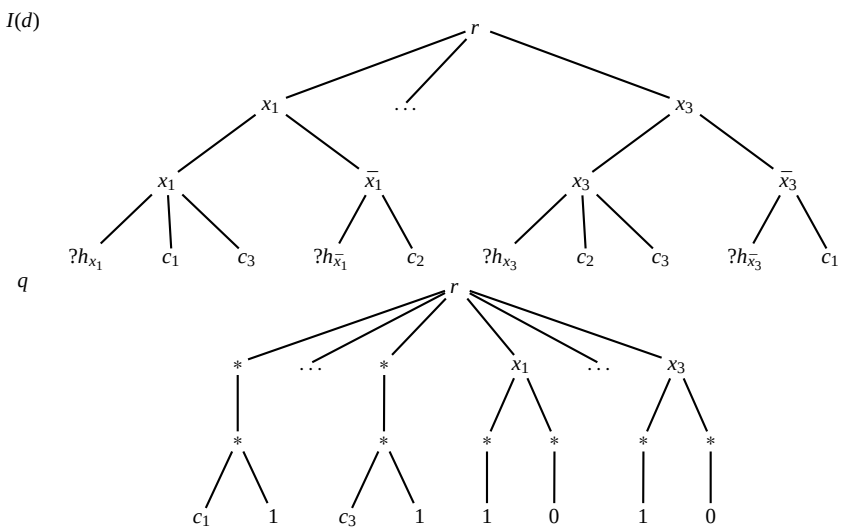


Figure 12: The construction of φ for Theorem 6

$\Leftarrow$ Conversely, let v be a valuation which satisfies φ. For each variable, let ω_x be $add(?h_x, 1)add(?h_{\bar{x}}, 0)$ if $v(x) = 1$ and $add(?h_x, 0)add(?h_{\bar{x}}, 1)$ otherwise. Consider the update $\omega_{x_1} \ldots \omega_{x_n}$, where the variables are $x_1 ... x_n$. One can verify that $\omega(I)$ satisfies the query.

PTIME (in data size): The proof uses again the notion of scenarios. Let us consider a document I and a query q. The number of scenarios is bounded by a polynomial function in the size of $|I|$. If we consider one scenario, this is satisfiable if, the set of subqueries to be matched by one function, could be matched with one message. This comes to testing that, for one function, the root of all subqueries to be matched in a scenario is labeled with the same label or with *. Observe that one does not need to consider those subqueries for a pattern node p with $(p', p) \in E_{//}$. The check for the remaining subqueries can be done in a polynomial time in $|q|$. There are polynomial many tests for one scenario (their number is at most the number of functions in the document) . So, the complexity stays polynomial in the size of the document. □

Proof of Theorem 8.

(sketch) We reduce the problem of implication of function dependencies (fd's in short) and inclusion depencies (ind's in short) for relational databases to the satisfaction problem for queries with negations. (Satisfaction can then easily be reduced to document satisfiability.)

We recall classical definition of dependencies in the relational model:

fd $I \models A_1 \ldots A_m \rightarrow B$ (where A_i and B are attributes) iff for each tuple $s, t \in I$, $s(A_i) = t(A_i)$ for each i implies $s(B) = t(B)$.

ind $I \models A_1 \ldots A_m \subseteq B_1 \ldots B_m$ (where A_i, B_i are attributes), if for each tuple r in I, there exists a tuple s in I such that for each i, $r(A_i) = s(B_i)$.

Consider a relation R over $A_1 ... A_m$. We can represent it naturally as a tree with root labeled by r, with a child labeled R, that has children labeled t (one per tuple), each with m children labeled $A_1, ..., A_m$, each with a child with the data value as label. One can construct queries as follows:

1. a Boolean query γ that tests whether the document d is indeed a representation of a relation R.
2. for each dependency χ (functional or inclusion), a Boolean query $q(\chi)$ that checks whether χ is satisfied.

The queries for a functional and inclusion dependencies are given in Figure 13. The construction of γ is ommited.

Let $\chi_1, ..., \chi_n$ and χ be functional or inclusion dependencies. Consider the query q consisting of a root labeled r, positive subpatterns for each χ_i, and γ, and a negative one for χ. Then one can prove that:

$$\chi_1 \wedge ... \wedge \chi_n \nvdash \chi \text{ iff } \exists d(d \models q)$$

Proof of Theorem 9 .

Consider the combined complexity.

LEMMA 5. *The problem of deciding, given a document and a no-join Boolean query, whether a function is useful, is* NP-COMPLETE *in the size of the query and the document.*

PROOF. (sketch)

NP: Assuming q is not satisfied, to show that $?f$ is a useful function, it suffices to exhibit an update sequence ω such that $\omega(I) \models q$ and that $\omega_{no\text{-}f}(I) \not\models q$. The test can be performed in PTIME. It remains to see if it suffices to consider a sequence of polynomial size. Suppose such an ω exists. We can take a minimum subsequence of ω so that $\omega(I) \models q$. By definition of q, there is one, say ω', that has no more updates than the size of q. Now it could be the case that the size of one update in ω' is huge. But the interesting part in it has to be small - about the size of q.

Hard: The proof is again by reduction of 3SAT. Let $\varphi = \wedge_{i \in [1..n]} C_i$ be such a formula. The corresponding instance is constructed as follows. For each C_i, let c_i be a distinct new label. The instance also uses functions h_j^0 and h_j^1 for each x_j. The root, labeled r, has one subtree t_j for each variable x_j and one other subtree t_c. The subtree t_j has a root labeled a and two subtrees, t_j^0, t_j^1 defined as follows:

- t_j^0 has root labeled a, one children labeled c_i for each C_i where $\bar{x_j}$ occurs, and two other subtrees: one consisting of a single node labeled 0; and one subtree $a[h_j^0]$.
- t_j^1 has root labeled a, one children labeled c_i for each C_i where x_j occurs, , and two other subtrees: one consisting of a single node labeled 1; and one subtree $a[h_j^1]$.

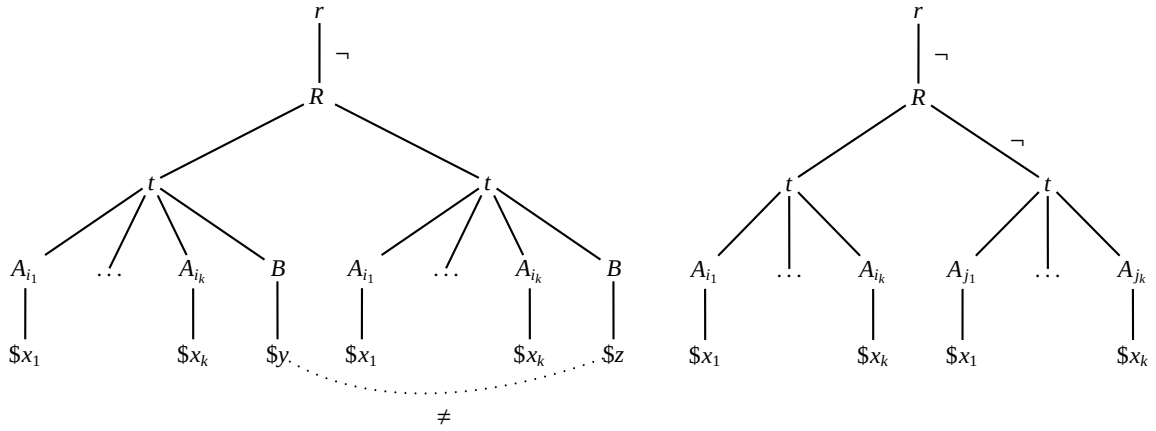


Figure 13: Queries for an fd (left) and an ind (right)

The subtree t_c is: $a[\ a[1\ a[1]]\ a[0\ a[h]]\ ]$; where h is the function for which we question the usefulness.

The query q is constructed as follows. It has a root r with one subtree q_i for each clause C_i plus a subtree q_c, defined as follows. Query q_i has a root labeled a and a unique child

$$a[\ c_i\ a[1]\ ]$$

The other child of the root of q is

$$a[\ a[1\ a[1]\ ]\ a[\ 0\ a[1]\ ]\ ]$$

The instance I and the query q are shown in Figure 14 for

$$\varphi = \underbrace{(x_1 \vee x_2 \vee \bar{x_3})}_{C_1} \wedge \underbrace{(\bar{x_1} \vee \bar{x_2} \vee x_3)}_{C_2} \wedge \underbrace{(x_1 \vee \bar{x_2} \vee x_3)}_{C_3}$$

One can show that φ is satisfiable iff h is useful.

Suppose φ is not satisfiable. Then every instance that satisfies $\wedge q_i$ also satisfies q_c, so h is useless.

Conversely, suppose that φ is satisfiable. Let ν be a valuation that satisfies φ. Consider the update ω that consists in sending a 1 to each h^1_j of $\nu(x_j) = 1$ and sending a 1 to h^0_j otherwise. Let $J = \omega(I)$. Then $J \not\models q$ whereas $add(h, 1)(J)$ does. Thus h is useful. □

Now consider the data complexity. To show it is PTIME, we introduce an algorithm that allows computing useful functions. As we will see, it runs in PTIME in the size of the data.

Given a document d and a query q, one can build an algorithm which is based on the *ScenariosSet* obtained with Algorithm 4. To decide the usefulness of a function $?f$, it searches for update sequences so that an $?f$-update is necessary in them for the document to satisfy the query.

The Lemma 7 provides the procedure for building minimal scenarios. To state it, we need to introduce some definition and notation. First, in a standard manner, q is *contained* in q', denoted $q \subseteq q'$, iff each document satisfying q also satisfies q'. We say that the pattern rooted at p is brought by a tree if that tree satisfies $[p]_q$ (defined at the beginning of the appendix). Given a scenario u and a function $?g$, the forest of patterns *brought by* $?g$ *in* u, denoted $broughtBy(g, u)$ is the set of $[p]_q$ such that for some query node p, $u(p) = ?g$. Observe that the semantics of $[p]_q$ is more complex when p is a pattern node such that $\exists p', (p', p) \in E_{//}$. In this case, the pattern rooted at p can be mapped no matter where in the tree (not necessarily at the root of the tree). We represent $[p]_q$ in this case as a disjunction of queries $\lfloor q \rfloor \vee //\lfloor q \rfloor$ (see also Figure 11).

To prove the correctness of the algorithm, we use the following lemmas.

The first lemma states that no useful update sequence is missed by the datalog program.

LEMMA 6. *An update sequence ω is useful iff there is a tuple $u \in ScenariosSet$, such that for each $p, u(p) \neq \perp$ there is an update $(u(p), \alpha) \in \omega$ such that $\alpha \models [p]$.*

PROOF. ⇒ Let ω be a useful update sequence for the query q and the instance I. Let $m\omega$ be a minimal useful update sequence extracted from ω. Then, there is a valuation ν from q to $m\omega(I)$. So let $L(\nu, I)$ be the set of query nodes p such that $\nu(p) \in m\omega(I) \setminus I$ and for the parent of p, denoted p': $\nu(p') \in I$. In the computation of the program $\delta_q(I)$, $\forall p \in L(\nu, I)$, there exists a node function n such that $p(n) = \frac{1}{2}$. And for all ancestor p' of p and ancestor of n, denoted n': $p'(n') < 1$. So, $relevant_p(n) = 1$ and there exists a tuple u such that $u(p) = \lambda(n)$. The way the u's were built ensures that one of these u's is compatible for all p.

⇐ Let be a sequence ω and u such that for each $p, u(p) \neq \perp$ there is an update $(u(p), \alpha)$ with the property: $\alpha \models [p]$. First, if $u(p) = ?f$ then $n \in \lambda^{-1}(?f), relevant_p(n) = 1$, so for all $p' \in \lceil p \rceil$ there exists $n' \in \lceil n \rceil, p'(n') = \frac{1}{2}$. It is easy to check that after the update with the update $(?f, \alpha)$ such that $\alpha \models [p]$, there exists a node n_1 (cousin of n) such that $p(n_1) = 1$. The way u's are built ensures that those updates give a valuation from p to $\omega(I)$.

□

Now, we can prove Lemma 7.

LEMMA 7. *Let I be an instance and q a query. A function $?f$ is useful for I and q iff there exists a scenario $u \in ScenariosSet$, $?f$ occurs in u and $q' \not\subseteq q$ where q' is obtained from the instance I by (i) removing each occurrence of $?f$ and (ii) replacing each occurrence of some function call $?g$ by*[5] *$broughtBy(g, u)$.*

PROOF. ⇒ Let $?f$ be a function such that there is no tuple u for which $?f$ is a value and $q_{I,u_{no}\text{-}f} \not\subseteq q$. If there is no tuple for which $?f$ is a value, $?f$ is useless. Otherwise, there are u's such that $?f$ is one of their values, but for those tuples

[5]Note that if $?g$ does not occur in u, then $broughtBy(g, u)$ is empty and $?g$ is simply removed like $?f$.

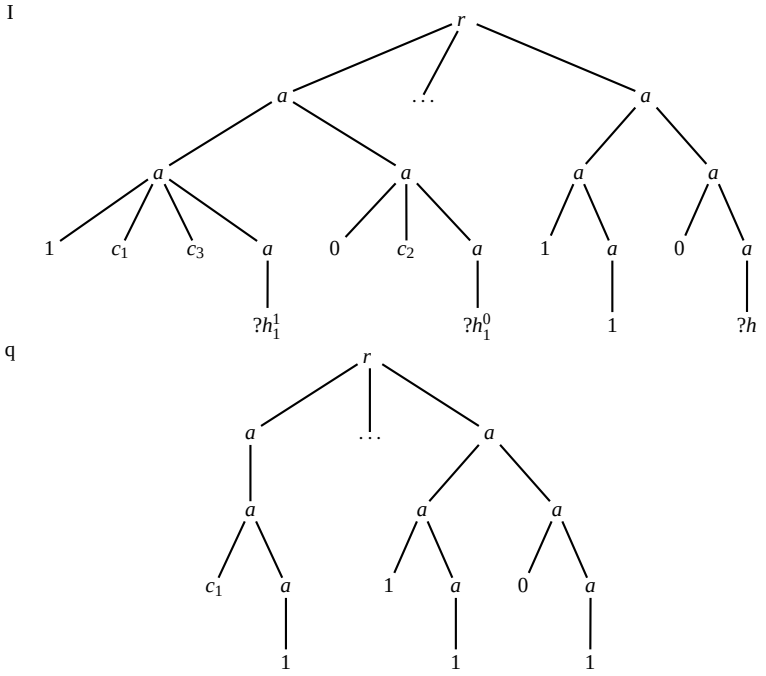


Figure 14: The construction of φ for Lemma 5

$q_{I,u_{no\text{-}f}} \subseteq q$. Let ω be a useful update sequence such that there is a tuple where $?f$ appears. Then $\omega_{no\text{-}f}(I) \models q_{I,u_{no\text{-}f}}$ and so $\omega_{no\text{-}f}(I) \models q$. Then $?f$ is useless.

$\Leftarrow$ Let be u such that $?f$ is one of its values and $q_{I,u_{no\text{-}f}} \not\subseteq q$. There exists an instance I' such that $I' \models q_{I,u_{no\text{-}f}}$ and $I' \not\models q$. There exists an update sequence ω such that $\omega(I) = I'$. This update sequence is not useful but it can be extended to a useful sequence with $?f$-messages. Indeed, $\forall p, u(p) \neq ?f \wedge u(p) \neq \perp$, there exists a $u(p)$-occur α such that $\alpha \models [p]$. By Lemma 6, it is enough to find messages coming from $?f$ that verify $[p], u(p) = ?f$ so that the update sequence ω become useful. The function $?f$ is then useful.

□

The data complexity of the usefulness decision problem is given by the lemma:

LEMMA 8. *The problem of deciding, given an instance and a no-join Boolean query, whether a function is useful, is* PTIME *in the size of I.*

PROOF. We use Lemma 7. We denote $|q|$ and $|I|$, the size of the query and the size of the document. The number of scenarios is polynomial in $|I|$. In order to evaluate the complexity in $|I|$ for the test $q_{I,u_{no\text{-}f}} \subseteq q$, we use the complexity result of G. Miklau and D. Suciu [22]. This result is valid for two queries without disjunction.

Remark that in $q_{I,u_{no\text{-}f}}$ there might be disjunctions. We can show that this is not a problem: we can expend $q_{I,u_{no\text{-}f}}$ in a set of no-disjunction queries, denoted S, such that: $\forall t, t \models q_{I,u_{no\text{-}f}}$ iff $\exists q_1 \in S$ such that: $t \models q_1$. So, it is obvious that $q_{I,u_{no\text{-}f}} \subseteq q$ iff $\forall q_1 \in S, q_1 \subseteq q$. The size of S is independent of $|I|$ and $\forall q_1 \in S, |q_1| \leq |q_{I,u_{no\text{-}f}}|$. Without loss of generality, we can assume that there is no disjunction in $q_{I,u_{no\text{-}f}}$.

The complexity of the algorithm given by Miklau and Suciu to check $q_1 \subseteq q_2$ is in $O(|q_1|\ |q_2|\ (w' + 2)^{c+1})$ where c is the number of couples in the relation $E_{//}$ of q_1 and w' is the size of the largest sequence of nodes of q_2 labeled $*$, compatible with the relation $E_{/}$ of q_2. In our case the two parameters previously mentioned are independent of $|I|$.

The size of $q_{I,u_{no\text{-}f}}$ is bounded by a polynomial function of $|I|$, so the complexity to check $q_{I,u_{no\text{-}f}} \subseteq q$ is in PTIME in $|I|$. This ends the demonstration. □