Incremental View Maintenance for Active Documents

Serge Abiteboul, Pierre Bourhis, Bogdan Marinoiu

Incremental View Maintenance for Active Documents[*]

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

July 23, 2007

Abstract

In this paper, we develop algorithmic datalog-based foundations for the incremental processing of tree-pattern queries over active documents, i.e. document with incoming streams of data. We define query satisfiability for such documents based on a logic with 3-values: "true", "false forever", and "false for now". Also, given an active document and a query, part of the document (and in particular, some incoming streams) may become irrelevant for the query of interest. We introduce an incremental algorithm for detecting such useless data and streams, essential for implementing garbage collection. We also provide complexity analysis for the problems we study.

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

1 Introduction

One of the essential components of Internet data management is the processing of (possibly very intensive) streams of data exchanged by data sources. In the spirit of ActiveXML [3], such processing is modeled here by a query posed to an active document, i.e., to a document that evolves by receiving incoming flows of XML data. The content of the document keeps changing monotonously because of the incoming data. The query (a tree-pattern query) keeps producing an XML flow as output. In this paper, we develop algorithmic datalog-based foundations for such an incremental query processing.

Our motivations come in particular from two very important settings:

Distributed monitoring When monitoring distributed applications, the monitored systems generate streams of alerts and the main role of the monitoring system is to process queries over these streams. The streams of alerts can be collected into an (active) document. The monitoring problem becomes an instance of the view maintenance problem over active documents. The role of the system is to incrementally maintain the view query.

Distributed query processing As stressed for instance in [5], in distributed query processing, the core local computation typically consists in receiving flows of data and evaluating a query over them.

To see an example of monitoring, suppose that we are interested in detecting the electronic orders such that delivery is performed more that t seconds after receiving payment. The monitoring system receives alerts for the payment (from

[*]This work is partially supported by ANR-06-MDCA-005 grant DocFlow.

[†]ENS Cachan.

[‡]Address: Parc Club Orsay Université, ZAC des vignes, 4, rue Jacques Monod - Bâtiment G, 91893 ORSAY Cedex

the banking system) and the delivery (from the warehouse system), which will be collected into an (active) document. A join based on order Id followed by a selection detects the slow deliveries. To see an example of distributed query processing, suppose that the query processor is performing a Holistic Twig join between posting lists distributed in a DHT. It receives streams of postings from different peers and performs a join over these streams.

Contribution We first consider Boolean tree-pattern queries over active documents including input streams. It is known that the evaluation of such queries can be expressed in monadic recursion-free datalog, so that it can be performed in linear time. A difference in our setting is that although the answer to the query (or to a subquery) is false, it may be the case that the query may become true in the future. So, we use 3-valued logic. In a temporal logic style, a fact is either true, false for now but with a chance to become true (if some appropriate data is received), or false forever. We show how to perform in linear-time this evaluation using a 3-valued datalog program and how to maintain it incrementally.

We extend the technique to non-Boolean tree-pattern queries, and to queries with join and disjunction. The algorithm runs in PTIME and can also use all the incremental datalog technology. We should observe that with very little more the problem becomes hard. This is the case, for instance, if some queries are allowed to be non-stream (i.e. classical functions that return a result and terminate) or if we introduce typing for the documents.

We also consider some form of (stream) garbage collection. To illustrate, suppose that the document consists of a collection of trees, one for each manager of a company, and that we want to output the name of managers such that their corresponding trees verify a certain pattern (e.g., managers that have already produced the yearly reports for all the projects they manage). As soon as the tree of a manager satisfies the pattern, we can output her name and "garbage collect" the corresponding tree that became useless. In particular, it is not necessary to continue processing the streams that are coming in that tree. Similarly, if the tree corresponding to a particular employee has no chance to ever meet the conditions, it can also be garbage-collected. We show that the problem is hard even in the simplest setting of Boolean-queries. We propose a technique that computes incrementally the nodes (so the functions that can be garbage-collected). This is also based on the 3-valued logic already mentioned.

The paper is organized as follows. The model is presented in Section 2. Boolean queries are considered in Section 3 and more complex queries in Section 4. The usefulness of functions is the topic of Section 5. The last section is a conclusion.

2 Model

In this section, we formalize the data structure, i.e., active documents in the style of ActiveXML [3], and the queries, namely tree-pattern queries with joins, that are considered in the paper.

We assume the existence of some infinite alphabets $\mathcal{I}$ of node identifiers, $\mathcal{L}$ of labels, $\mathcal{N}$ of document names and $\mathcal{F}$ of function names. We do not distinguish here between data and labels, i.e., our notion of label is meant to capture the notions of XML element label and XML PCDATA. 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.

Definition 1 (Active data and document) *An* (active) tree *is a pair* (t, λ) *where (1)* t *is a binary relation that is a finite tree*[1] *with nodes* $nodes(t) \subset \mathcal{I}$ *; (2) a labeling function* λ *over* $nodes(t)$ *with values in* $\mathcal{L} \cup \mathcal{F}$*; and (3) only*

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

leaves are labeled by values in $\mathcal{F}$. An (active) forest *is a set of (active) trees. An* (active) document *is an (active) tree whose root is in $\mathcal{L}$.*

The trees I, t, J in Figure 1 are examples of active trees. A subtlety is the distinction between an active tree and an active document. A tree may yield a forest (since a function call may return a stream of trees) whereas a document always remains a single tree.

Two active trees (t, λ), (t', λ') are *isomorphic* if they differ in their node identifiers only. The two trees are interchangeable.

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 ActiveXML terminology). Such nodes represent subscriptions to Web services and are meant to receive some data. Note that (3) is a restriction from ActiveXML, where trees underneath function calls denote the arguments of the call. We assume here that the calls have been performed and a symbol $!f$ is just meant as the place holder for receiving the result of this call.

A tree without any function call is called an *XML tree* (by opposition to an ActiveXML tree). A function returning only such trees is called an *XML function.* To simplify, we will mostly focus on XML functions here.

Definition 2 (Schema and instance) *A* schema (S, F, τ, Σ) *(or S when the other components are understood) consists of a finite set S of document names, a finite set F of function names, a typing τ (for the documents and functions) and a set Σ of constraints. Let (S, F) be a schema. An* instance *I of (S, F) is a function that maps each d in S to a document $I(d)$, such that each function name occurring in $I(d)$ is in F.*

In this paper, we will (mostly) ignore the types and the constraints. When S consists of a single document name d, we talk about the schema d.

A function in such an active document brings in updates to the document. In the present paper, we consider that the incoming flow of updates consists only of insertions. So, the content of the document is monotonously increasing. Clearly, it would be also interesting to consider non monotone changes. This is left for future research.

An *update* for (S, F) is an expression $add(!f, \alpha)$ where $!f \in F$ and α is a data tree with functions only in F. Let I be an instance over (S, F) and $add(!f, \alpha)$ an update of (S, F). When the result α does not contain any function node, the update is called an *XML update.* The *result* of applying $add(!f, \alpha)$ to I, denoted $add(!f, \alpha)(I)$ is the instance obtained from I by adding, for each node n labeled $!f$, a fresh copy[2] $h_n(\alpha)$ of α, as a sibling of n. The instance obtained from I by applying a sequence w of updates is denoted $w(I)$.

For instance, for I, t, J as in Figure 1, $J = add(!f, t)(I)$.

We consider monotone queries over a single document, more precisely, tree-pattern queries. A generalization to multi-document queries may be found in [2].

Definition 3 (Tree-pattern queries) *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; and*
- *the labeling function λ maps each node in $nodes(q)$ to $\mathcal{L} \cup \{*\}$;*
- *the projection π is a set of nodes in $nodes(q)$.*

where $nodes(q)$ is the set of nodes occurring in $E_/$ or $E_{//}$ and $*$ is a wild card that matches any label.

If the arity of π is 0, the query is said to be *Boolean.* Its result is then either the empty set (false) or the set containing the empty tuple (true). Examples of tree-pattern queries are

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

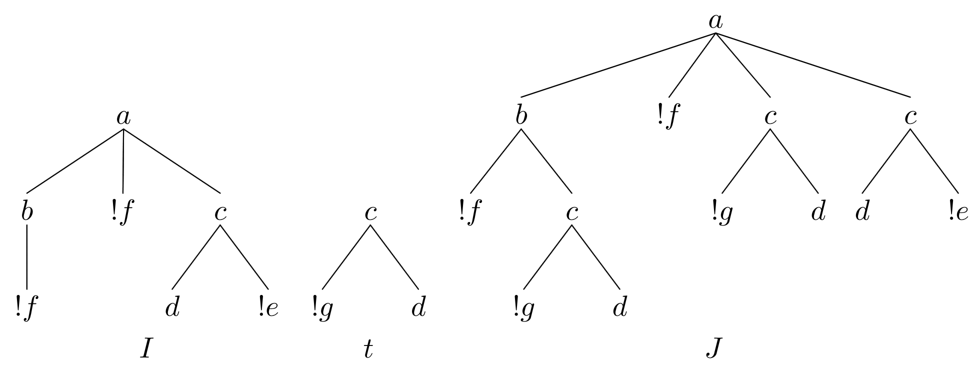

Figure 1: Example for active trees and updates

given in Figure 2. The nodes whose labels are requested in the result are marked with a "+" and the edges in $E_{//}$ are indicated by || on top of the target node. Query q_1 and q_2 are examples of Boolean queries whereas the result of q_3 is a binary relation.

The semantics of queries is defined as follows.

Definition 4 (Valuation and result) *Let $q = (E_/, E_{//}, \lambda, \pi)$ be a tree-pattern query and (t', λ') a document. A* valuation *ν (from q to (t', λ')) is a mapping from $nodes(q)$ to $nodes(t)$ satisfying the following properties:*

- *Root-preserving: $\nu(root(q)) = root(t)$.*
- *Label-preserving: For each $p \in nodes(q)$, $\lambda(p) = *$ or $\lambda(p) = \lambda'(\nu(p))$.*
- *Parent-preserving. For each $(p, p') \in E_/$, $\nu(p)$ is a parent of $\nu(p')$ in t.*
- *Descendant-preserving. For each $(p, p') \in E_{//}$, $\nu(p)$ is an ancestor of $\nu(p')$ in t.*

The result *$q(t, \lambda)$ is the relation $\{\lambda'(\nu(\pi)) \mid \nu \text{ a valuation}\}$.*

We say that an instance $I(d)$ of a document *verifies* a Boolean tree-pattern query q, denoted by $I(d) \models q$, if there exists a valuation from q to $I(d)$. Using standard logic notation, we will denote the fact that a tuple u is in the answer by $I(d) \models q(u)$.

Two queries are *equivalent* if they yield the same result for each document.

We will also consider disjunctions of tree-pattern queries (denoted with $\vee$) with the standard obvious semantics.

In the remaining of the paper, we will use the following notation that can be ignored for a first reading of the paper.

Notation Let q be a query and p a node in q. Then

- ${}^\bullet p$ is the parent of p.
- $p^\bullet$ is the set of the children of p.
- $\lfloor p \rfloor_q$ is the query rooted at p.
- $\lceil p \rceil_q$ is the query corresponding to the path from the root of q to p and no other node.

And similarly for $\lfloor n \rfloor$, $\lceil n \rceil$ for a data tree t and a node n in t.

Also $//q$ is the query such that (i) its root is labeled $*$ and has a single outgoing $//$-edge; and (ii) the subtree rooted at the child of the root is q.

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 $[p]_q = \lfloor p \rfloor_q \vee //(\lfloor p \rfloor_q)$. $\Box$

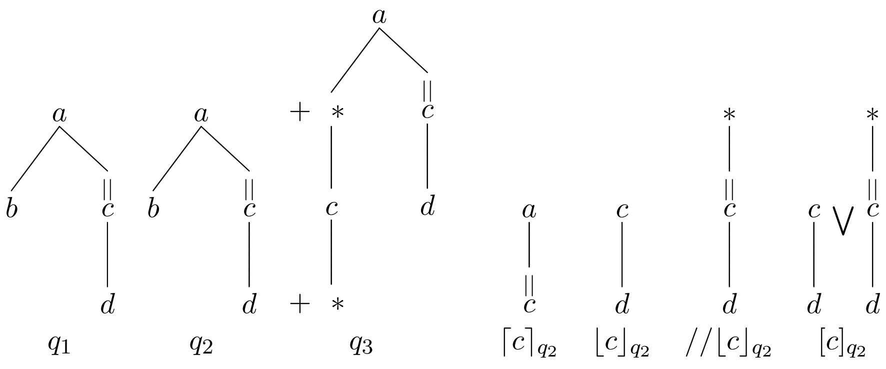

Figure 2: Examples of queries

When Query q is understood, they are respectively denoted $\lceil p\rceil, \lfloor p\rfloor, [p]$.

For q_2 as in Figure 2, $\lceil c\rceil_{q2}$, $\lfloor c\rfloor_{q2}$, $//\lfloor c\rfloor_{q2}$ and $[c]_{q2}$ are shown in Figure 2.

Possibly Satisfiable We will use a 3-valued logic that permits distinguishing between what is false forever and what is false but still has a chance to become true in the future. In this context, we want to refine the notion of satisfiability. An instance $I(d)$ *possibly satisfies* a Boolean query q if there exists a sequence w of updates such that $w(I) \models q$. We denote it by $I(d) \models_\diamond q$. Because we consider only monotone queries and add-only updates, for a Boolean query q, one of the following three cases occurs:

true (1) I satisfies q and no matter what update happens, it will always do. $(I \models q)$

maybe $(\frac{1}{2})$ I does not satisfy q but it possibly will in the future. $(I \not\models q \wedge I \models_\diamond q)$

false (0) I does not satisfy q and it will never do. $(I \not\models q \wedge I \not\models_\diamond q)$

In this 3-valued logic, a fact A is either true, false for now and possibly true in the future or false forever, i.e., respectively have value 1, $\frac{1}{2}, 0$. The semantics of logical operators in this setting correspond to the following truth values: for X, Y in $\{1, \frac{1}{2}, 0\}$, $X \wedge Y = min(X, Y)$, $X \vee Y = max(X, Y)$ and $\neg X = 1 - X$.

We can also extend the notion of *result* to a query. A tuple u is *surely* in the result if $u \in q(I)$. It is *possibly eventually* in the result if for some ω, $u \in q(\omega(I))$. As standard in logic, we will also use the notation $I(d) \models_\diamond q(u)$, to formally state that u is possibly eventually in the result of q.

We also extend the notion of (possible) satisfiability to forests in the obvious way. In particular, a forest *(possibly) satisfies* a query q if some tree in it (possibly) satisfies q.

In Figure 3, $I_1 \models q$, $I_2 \not\models_\diamond q$ and $I_3 \models_\diamond q$. For the last one, some c value returned by f_1 may turn satisfiability of q to true.

3 Boolean query evaluation

In this section, we consider the evaluation of Boolean tree-pattern queries as introduced in [18]. Such queries can be evaluated in a standard manner, e.g., with a regular tree language (see [12]) or a monadic datalog program (see [4]). The relation between Regular Tree Languages and Monadic Datalog is shown in [14]. Since we are concerned here with incremental maintenance, we use datalog so that we can benefit from the incremental datalog technology overviewed in [4]. We will mention in conclusion how this can be indeed implemented using an XQuery processor.

To simplify, we assume that all streams return "static" data, i.e., data not containing function

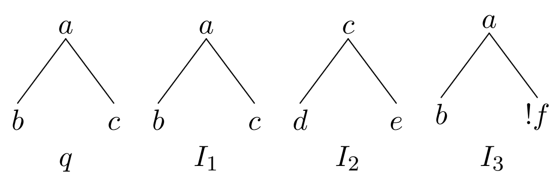

Figure 3: a query q and three documents : I_1, I_2, I_3

calls. In other words, we consider that all functions are XML functions (and not ActiveXML). We will briefly remove this extension further.

Following the formalism described in [14], we will assume the availability of the following relations: base relations $par(x, y)$ (for parent), $anc(x, y)$ (for ancestor), $root(x)$ (for root), $lab_a(x)$ (for each label a), $lab_*(x)$ (the union of all lab_a relations). We also use a relation $fun(n)$ that indicates that n is a function node.

Several optimization techniques have been proposed for the incremental evaluation of the Datalog programs. The most important are Query-Subquery (QSQ) presented in [22] and Magic Set described in [10]. These techniques have been adapted to our context. Consider a query q consisting of a root and two sub-branches the left one rooted at p_1 and the right one rooted at p_2. If the left branch of the query does not match some data, the Magic Set rewriting blocks the evaluation of the right branch of the query. This blocking occurs even if there is some hope that this left branch will be matched later, when new data will arrive. The 3-valued logic helps us overcome this difficulty. Indeed, our technique may be seen as an extension of Magic Set evaluation to this 3-valued setting.

For a given query q, we construct a 3-valued non-recursive datalog program δ_q that computes, in an efficient manner, whether the query is satisfied by the input document. For instance, for the query q of Figure 3, the datalog program, is given in Algorithm 1. We evaluate the query in 3-valued logic, i.e., we evaluate the true subformulas but also the $\frac{1}{2}$ ones. For instance, $\delta_q(I_1)$ is 1, $\delta_q(I_2)$ is 0; and $\delta_q(I_3)$ is $\frac{1}{2}$.

As in deductive database, our algorithm assumes some ordering of the query predicates, that we did not have thus far. So let $\preceq$ be a total ordering of the nodes of q. A node is the *benjamin* of some node p if it is the least child of p for$\preceq$. The set of benjamins is denoted $Benjamin_{\preceq}(q)$. When p is a child and not a benjamin, the *cadet* of p is the greatest sibling of p that is less than p. The cadet of p is denoted $cadet_{\preceq}(p)$.

Given q, Algorithm 2 (*PossibleSat*) constructs a program δ_q. One can prove:

Theorem 1 *For each query q and each instance I,*

- $\delta_q(I) = 1$ *iff* $I \models q$;
- $\delta_q(I) = \frac{1}{2}$ *iff* $I \not\models q$ *and* $I \models_\diamond q$;
- $\delta_q(I) = 0$ *iff* $I \not\models_\diamond q$.

It is shown in [14] that the evaluation of Boolean tree-pattern queries is in PTIME in the size of the data and the query. We can prove a similar result for the computation of possible satisfaction.

Theorem 2 *Possible satisfaction of Boolean tree-pattern queries is in* $O(|I|\ |q|)$*, so in* PTIME *in the data and the query.*

It is important to note that the technique presented here also provides *incremental maintenance.* The proof is by translating the 3 value-logic programs in Datalog programs with only two Boolean values: true and *false.* The reader interested in more details on the incremental maintenance of Datalog is kindly invited to consult [22].

4 More general queries

In this section, we consider more general queries than the ones presented in Section 3.

Some extensions can be obtained easily:

```
begin
    $q() \leftarrow a_1(n)$
    $\widehat{a}(n) \leftarrow root(n), lab_a(n)$
    $\widehat{b}(n') \leftarrow \widehat{a}(n), par(n,n'), lab_b(n')$
    $\widehat{b}(n') \leftarrow \widehat{a}(n), par(n,n'), fun(n'), \frac{1}{2}$
    $\widehat{c}(n') \leftarrow \widehat{a}(n), par(n,n'), lab_c(n'), (b(n'') \geq \frac{1}{2}), par(n',n'')$
    $\widehat{c}(n') \leftarrow \widehat{a}(n), par(n,n'), fun(n'), (b(n'') \geq \frac{1}{2}), par(n',n''), \frac{1}{2}$
    $b(n) \leftarrow \widehat{b}(n)$
    $b(n) \leftarrow \widehat{b}(n), fun(n)$
    $c(n) \leftarrow \widehat{c}(n)$
    $c(n) \leftarrow \widehat{c}(n), fun(n)$
    $a(n) \leftarrow b(n'), c(n''), \widehat{a}(n), par(n,n'), par(n,n'')$
end
```

Algorithm 1: $\delta|_q$ for the example query q

Disjunction One can consider queries with disjunction. More precisely, one could introduce some particular nodes that specify that "one of the subtree-patterns has to match". The algorithm from the previous section can be easily modified to carry such tests since datalog does capture disjunction. Observe that the algorithm still runs in linear time. Since this is straightforward, this extension will not be considered in more detail.

Terminating functions Suppose some functions terminate, e.g., by sending and *end-of-stream.* The algorithm from the previous section can also be used after first eliminating the functions that already terminated. So Boolean query evaluation remains in PTIME. Note that the incremental evaluation becomes a bit tricky because some facts that have already been inferred are invalidated. This aspect will not be pursued here.

Active result We have assumed so far that the streams return "static" data (i.e., XML documents). We could also consider that some data returned by a stream is itself an active document. It is straightforward to extend the algorithms for such setting.

We next consider non-Boolean queries, then queries with joins that rely on such non-Boolean queries.

Non-Boolean queries The algorithm from the previous section can be modified to work with non-Boolean queries, i.e., queries returning tuples of values. The main difference is that relation p in the algorithm should keep the data that have to be returned, i.e., the labels in our formalism. As we will see, this data may still be unknown when the fact is only possible with missing parts expected from function calls. We use a particular symbol "?", for such unknown data entry. Tuples including such unknown data will always be only possible (i.e., the truth value of such tuples will always be $\leq \frac{1}{2}$). In Figure 4, the answer of q for d_1 is (b,c) with the truth value 1, that for d_2 is $(b,?)$ with truth value $\frac{1}{2}$ and that for d_3 is (b,c) with truth value $\frac{1}{2}$.

We next extend the notion of satisfiability and possible-satisfiability to tuples possibly with unknowns. To do that we use the following notation. Recall that a result is a tuple over a set of (query) nodes. Let u and u' be two tuples over the same set $\mathbf{N}$ of attributes with values in $\mathcal{L} \cup \{?\}$. Then $u \leq u'$ iff for each $n \in N$, $u(n) = u'(n)$ or $u(n) = ?$.

```
Data: a tree-pattern query $q$
Result: the datalog program $\delta = PossibleSat(q)$
begin
    for $p = root(q)$ of label $a$ do
        Top-down evaluation;
        $\delta \mathrel{+}:= \ q() \leftarrow p(n);$
        $\delta \mathrel{+}:= \ \widehat{p} \leftarrow root(n), lab_a(n);$
        Bottom-up evaluation;
        $\delta \mathrel{+}:= \ p(n) \leftarrow \widehat{p}(n), p_{i_1}(n_1), \ldots, p_{j_1}(n'_1), \ldots, anc(n, n'_1), \ldots$ ;
                where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$;
    foreach $p' \in nodes(q)$ and child $p$ of $p'$ and of label $b$ do
        Top-down evaluation;
        if $p \in Benjamin_{\preceq}$ then
            if the arc between them is direct child then $E_/$
                $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), lab_b(n);$
                $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), fun(n), \frac{1}{2};$
            else $E_{//}$
                $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), lab_b(n);$
                $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), fun(n), \frac{1}{2};$
        else
            if the arc between them is direct child then $E_/$
                if the arc between $p'$ and $p'' = cadet_{\prec}(p)$ is direct child then
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), lab_b(n), (p''(n'') \geq \frac{1}{2}), par(n', n'')$
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), fun(n), (p''(n'') \geq \frac{1}{2}), par(n', n''), \frac{1}{2};$
                else
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), lab_b(n), (p''(n'') \geq \frac{1}{2}), anc(n', n'')$
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), par(n', n), fun(n), (p''(n'') \geq \frac{1}{2}), anc(n', n''), \frac{1}{2};$
            else $E_{//}$
                if the arc between $p'$ and $p''$=$cadet_{\prec}(p)$, is direct child then
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), lab_b(n), (p''(n'') \geq \frac{1}{2}), par(n', n'');$
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), fun(n), (p''(n'') \geq \frac{1}{2}), par(n', n'');$
                else
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), lab_b(n), (p''(n'') \geq \frac{1}{2}), anc(n', n'');$
                    $\delta \mathrel{+}:= \ \widehat{p}(n) \leftarrow \widehat{p'}(n'), anc(n', n), fun(n), \frac{1}{2}, (p''(n'') \geq \frac{1}{2}), anc(n', n'');$
        Bottom-up evaluation;
        $\delta \mathrel{+}:= \ p(n) \leftarrow \widehat{p}(n), p_{i_1}(n_1), \ldots, p_{j_1}(n'_1), \ldots, par(n, n_1), \ldots, anc(n, n'_1), \ldots$ ;
                where $(p, p_{i_x}) \in E_/$, $(p, p_{j_y}) \in E_{//}$;
        $\delta \mathrel{+}:= p(n) \leftarrow \widehat{p}(n), fun(n)$ ;
end
```

Algorithm 2: PossibleSat for Boolean queries

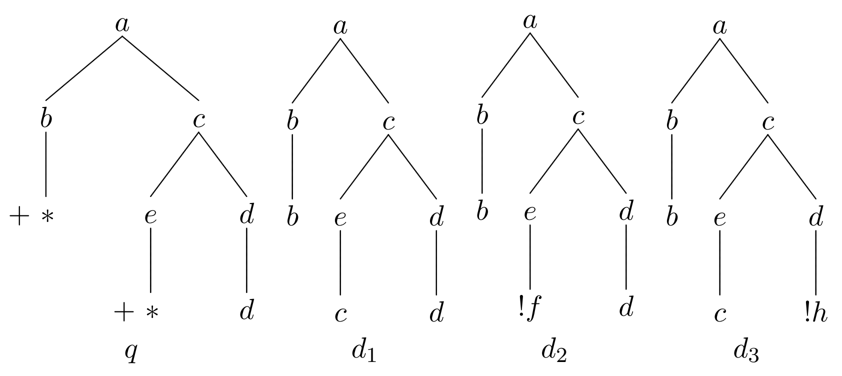

Figure 4: Example of non-Boolean query q and three documents d_1, d_2 and d_3

Now, the notions of satisfiability and possible-satisfiability are extended for tuples with unknowns as follows:

- $I \models q(u)$ iff $u \in q(d)$.
- $I \models_\diamond q(u)$ iff $\forall u', u \leq u'$, $I \models_\diamond q(u')$.

To evaluate a query q, we use again a non-recursive datalog program δ_q. The result of the evaluation of δ_q on input I is denoted by $\delta_q(I) = (\delta_q^1(I), \delta_q^{\frac{1}{2}}(I))$ where $\delta_q^i(I)$ is the set of facts with truth value i. (Note that a fact is false iff it is in neither.) The construction of δ_q is omitted as well as the analog of Theorem 1 that states its correctness.

Because of datalog, we now know that we can evaluate satisfaction and possible-satisfaction in PTIME in the size of the document. Observe however that the program is not monadic anymore, so it does not have to run in linear time. Also observe that again incremental evaluation is feasible.

Joins We now extend the query language with joins. Joins are essential in such settings because they allow matching information coming from different streams. Also, they allow expressing temporal properties such as an electronic delivery arrives less than 4 seconds after payment. To simplify we consider here only equality joins, but other kinds of joins can be processed similarly.

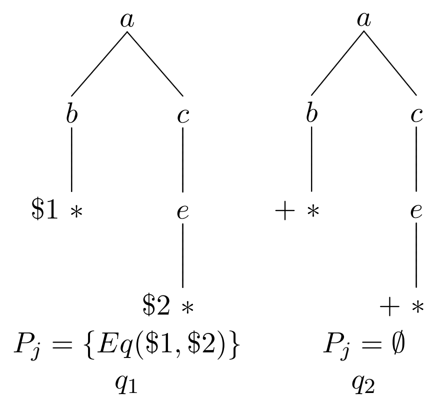

Figure 5: Example of tpj queries

Formally, joins are defined as follows.

Definition 5 (Tree-pattern-with-join query) *A* tree-pattern-with-join *(tpj in short) query is a pair (q, P_j) where q is a tree-pattern query and P_j (the join constraints) is a finite set of expressions of the form $Eq(p, p'')$, where p, p' are nodes in q. The notion of valuation is extended by requiring that, for each $Eq(p, p')$ in P_j, $\lambda(\nu(p)) = \lambda(\nu(p'))$.*

Query q_1 in Figure 5 is an example tpj query.

Intuitively, a node that has to be joined has to be carried along until it is joined, just like a node that has to be returned. So unsurprisingly, this will use the non-Boolean queries as a basis. Let q be a tpj query. Let P_j be its join constraint and

P_j* the reflexive, symmetric, transitive closure of P_j. (The need for this will become clear further.) Consider q' obtained from q by ignoring the join conditions but keeping in the result, the values of the join nodes. (For instance, for $q = q_1$ as in Figure 5, $q' = q_2$ is as in the same figure.) Then one can verify that

$$q = \pi_V(\sigma_{\wedge P_j*}(q'))$$

where π_V is the projection on the result nodes. To be precise, we also have to specify the meaning of selection conditions in presence of "?". For this, we state that the truth value of $=(?, a)$ is $\frac{1}{2}$ for each label a.

We next explain why $\wedge P_j$ does not suffice and we have to consider $\wedge P_j*$. To see that, consider the example in Figure 6 and the tuple $(a, ?, ?, b)$. It does satisfy P_j with truth value $\frac{1}{2}$. This is incorrect since it has no chance whatsoever to eventually succeed. The P_j* test rules it out.

By the construction for non-Boolean queries, we obtain a non-recursive datalog program for q'. It now becomes straightforward to obtain one for q, so the evaluation is in PTIME.

Remark: Observe that the computation that is obtained is rather inefficient. We carry the join values more than needed. This may be improved as follows. Consider $\tilde{P_j}$, the reflexive, transitive, symmetric closure of P_j. For each p, let $[p]$ be the equivalence class of the $\tilde{P_j}$ relation containing p. Let $rep([p])$ be some arbitrary representative for each class. For each query node p, we need only to keep in the schema of relation p:

> $rep([p'])$ for some p' if (i) some result node in the subtree rooted at p is in $[p']$ or (ii) some node in $[p']$ and in the subtree rooted at p joins with some node outside this subtree.

□

Non-stream functions The non-stream functions are classical service calls that return a result and terminate. In contrast with Theorem 2 for stream functions, possible-satisfaction is hard in this setting.

Theorem 3 *In presence of non-stream functions, possible-satisfaction of a Boolean tree-pattern query is NP-complete in the size of the query.*

Proof:

NP: To show that d possibly satisfies q, it suffices to exhibit an update sequence ω such that $\omega(d) \models q$. The test can be performed in PTIME. It remains to see that it suffices to consider a sequence of polynomial size. Suppose such an ω exists. We can take a minimum subsequence of ω so that $\omega(d) \models q$. By definition of q, there is one, say ω', that has no more updates than q itself. 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 (again by definition of q).

Hard: The proof is by reduction of 3SAT [13]. Let $\varphi = \wedge_{i\in[1..n]} C_i$ be a 3SAT formula.

From φ, we construct an instance of the possible-satisfaction 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 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 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:

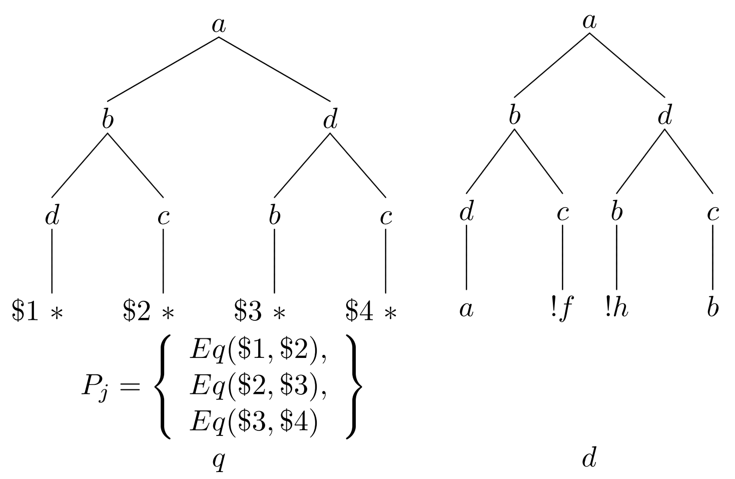
a
b
d
d
c
b
c
$1 *
$2 *
$3 *
$4 *
$P_j = \left\{ \begin{array}{l} Eq(\$1,\$2), \\ Eq(\$2,\$3), \\ Eq(\$3,\$4) \end{array} \right\}$
q
a
b
d
d
c
b
c
a
!f
!h
b
d

Figure 6: Example of tpj query and a document d

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

One can show that I_φ possibly-satisfies q_φ iff φ is satisfiable. □

Queries over typed documents When a typing may be imposed on the document, the problem is also hard.

Theorem 4 *Possible-satisfaction of a Boolean tree-pattern query for a typed document is NP-hard in the size of the query when the type of the document is specified by a tree automata.*

Proof: The proof is by reduction of 3SAT and is a variation of the proof of Theorem 3. The function h_x, $h_{\bar{x}}$ now are stream functions, but the type of the document prevents each of them from returning both a 0 and a 1. □

Note that typing may be introduced in a different manner by typing the return value of functions. Suppose to illustrate that we have a query q of root r with for unique subtree, a query q_1. Suppose also that the document d consists of a root r with as a single child a function $!f$. In general, we have to assume that $!f$ can bring some data so that q is possibly satisfiable. Now if we assume that the result of $!f$ obeys some type constraint, say q_0, the situation is different. Query q is then possibly satisfiable if $q_0 \wedge q_1$ is satisfiable. Observe that depending on the type language this may be expensive to check.

5 Useless functions

In this section, we present a technique for deciding whether parts of a document (in particular function nodes) are useful for the maintenance of a particular view. In practice, knowing that a subtree is useless is an important information. If this is the case, a subtree can be garbage collected. Also, a stream that brings data in such a useless part of a document is useless and it can be closed.

The main novel concept considered in the present section is that of *useful* function. Since we are in a monotonous context, note that if a Boolean query q is already satisfied, then no function is useful. This is also true if $I(d) \not\models_\diamond q$. An interesting example is the one of document d and query q in Figure 7. Note that $!h_2$ is useless because its d parent can neither match b nor c. On the other hand, $!f$ is useless because some cousin already matches all it can match and $!g$ is use-

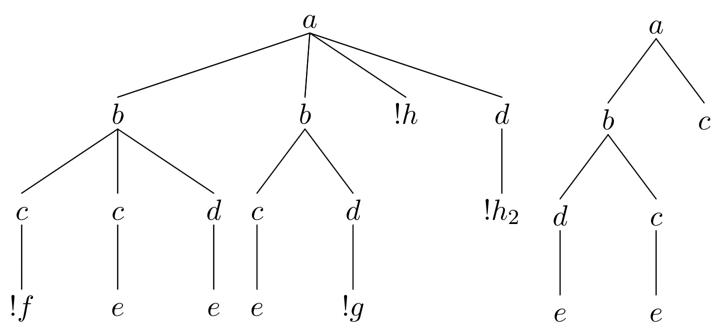

Figure 7: A document d and a query q

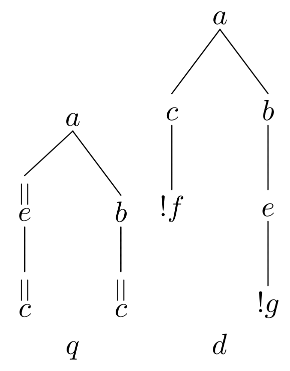

Figure 8: Example of hard usefulness

less because some node on the same level in the tree already matches all it can match. Now $!h$ is clearly useful since it can bring c child of the query's root.

The example in Figure 8 will illustrate the difficulty of the problem. Consider the sequence $\omega = (!f, e[c]), (!g, c)$ that yields to a document satisfying the query. A hasty analysis would lead someone to believe that $!f$ is useful because it could bring data that match a part of the query pattern. 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. Indeed, one can show more generally that $!f$ is useless for the document and the query in Figure 8.

More formally, for a sequence ω and a function $!f$, let $\omega_{no\text{-}f}$ denote the sequence obtained from ω by removing all $!f$-insertions. Now, we have:

Definition 6 *Let q be a query, $I(d)$ an instance, $I(d) \models_\diamond q$ and $I(d) \not\models q$. A function $!f$ is said to be* useless *for q and $I(d)$ iff $\forall$ ω update sequence, $\omega(I(d)) \models q$ iff $\omega_{no\text{-}f}(I(d)) \models q$.*

Intuitively, a function is useless if we can ignore its output and this does not change the satisfiability of the query. Note that in particular, this implies that, in a *useful* sequence ω of updates (i.e., a sequence that transforms a non-satisfying instance d into an instance satisfying q), we can erase messages brought by $!f$ and still obtain a useful sequence.

Unfortunately, deciding whether a function is useful is a hard problem as shown by the following theorem.

Theorem 5 *The problem of deciding, given an instance and a query, whether a function is useful, is NP-complete in the size of the query.*

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 $\omega(d) \models q$ and that $\omega_{no\text{-}f}(d) \not\models q$. 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 $\omega(d) \models q$. 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 $a[h_j^1]$.

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 9 for

$$\varphi = (x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee \bar{x_2} \vee x_3)$$

One can show that φ is satisfiable iff h is useful. $\square$

We next introduce an algorithm that allows computing useful functions. As we will see, it runs in PTIME in the size of the data (and EXPTIME in the size of the query).

Given a document d and a query q, the algorithm works as follows. 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. For that, we introduce the notion of *scenario*, that is a set of pairs (function call,query node) such that if each of function calls brings up messages satisfying the corresponding tree-pattern query (queries), the query q is satisfied. More precisely, a scenario is a tuple u over $nodes(q)$ with values in the set of function calls of d union $\{\triangle\}$. The meaning of $u(a) = !g$ is that $!g$ brings some data that matches the subquery rooted at a. The meaning of the $u(a) = \triangle$ is that a is irrelevant (because this subquery is already matched in this particular context).

The datalog program, namely *Useful*(q), computes *ScenarioSet*, a particular set of scenarios. This can be done relatively easily: after computing possible satisfiability, we record what functions could bring subqueries. The size of a relation corresponding to a node p in the query (relation noted $\tilde{p}$) is the size of the subquery rooted at p. The details are omitted. By construction, each possible scenario will be in *ScenarioSet*, i.e., this set provides a sufficient condition for usefulness. However, some of these scenarios may involve calls to functions that are not useful, so the condition it provides is not necessary for usefulness. This will be for instance the case for the example of Figure 8. The next theorem provides the procedure for building minimal scenarios from the ones delivered by Algorithm 3. 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'. Also 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 subqueries $\lfloor a \rfloor_q$ such that for some query node a, $u(a) = !g$.

Theorem 6 *Let I be an instance and q a query. A function $!f$ is useful for I and q iff there exists a scenario $u \in$ ScenarioSet, $!f$ occurs in u and*

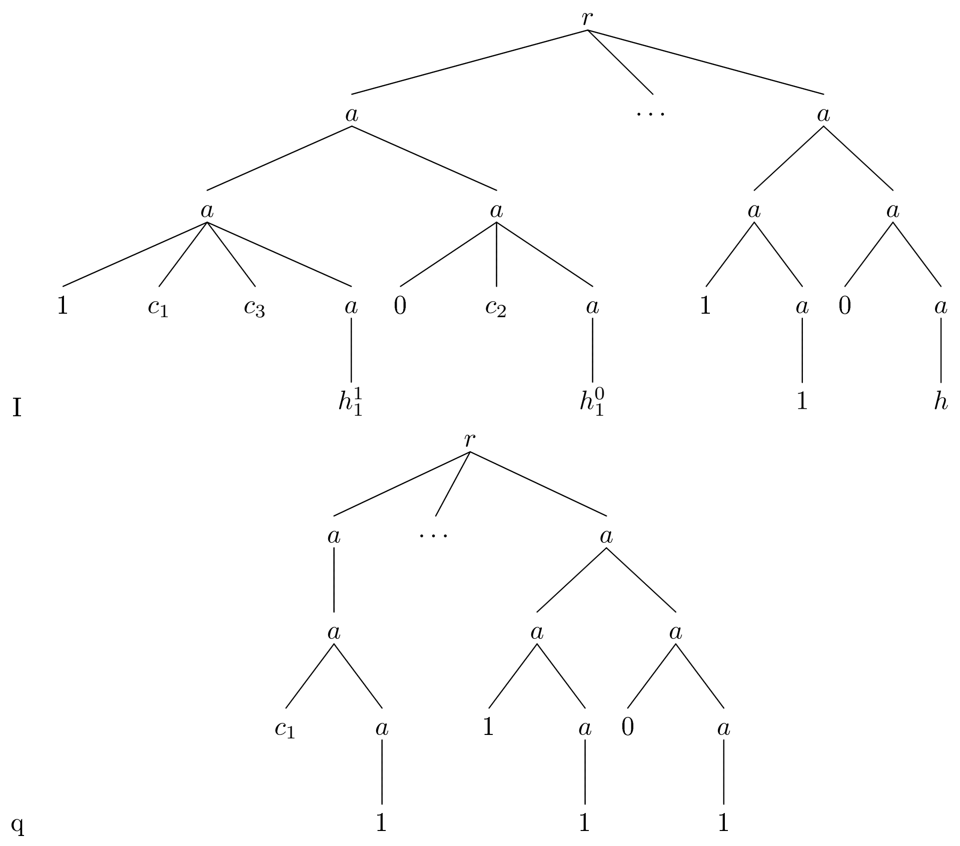

Figure 9: The construction for Theorem 5 of ϕ

```
Data: a tree-pattern query $q$, relations $p$ of $EvalSat(q)$
Result: the datalog program $\delta = Useful(q)$
begin
    for $p = root(q)$ of label $a$ do
        Top-down evaluation
        $\delta \mathrel{+}:= \; ScenarioSet(u) \leftarrow \tilde{p}(n,u)$
        $\delta \mathrel{+}:= \; relevant_p(n) \leftarrow root(n), p(n)$
        Bottom-up evaluation
        $\delta \mathrel{+}:= \; \tilde{p}(n,\triangle,u_1,\cdots,u'_1,\cdots) \leftarrow relevant_p(n), \tilde{p}_{i_1}(n_1,u_1),\cdots,\tilde{p}_{j_1}(n'_1,u'_1),\cdots,$
            $par(n,n_1),\cdots,anc(n,n'_1),\cdots$
                where $(p,p_{i_x}) \in E_/$, $(p,p_{j_y}) \in E_{//}$
    foreach $p' \in nodes(q)$ and child $p$ of $p'$ of label $b$ do
        Top-down evaluation
        if the arc between them is direct child then $E_/$
            $\delta \mathrel{+}:= \; relevant_p(n) \leftarrow relevant_{p'}(n'), par(n',n), (p(n) = \frac{1}{2}),$
                $\nexists n''((p(n'') = 1), par(n',n''))$
        else $E_{//}$
            $\delta \mathrel{+}:= \; relevant_p(n) \leftarrow relevant_{p'}(n'), anc(n',n), (p(n) = \frac{1}{2}),$
                $\nexists n''((p(n'') = 1), anc(n',n''))$
        Bottom-up evaluation
        $\delta \mathrel{+}:= \; \tilde{p}(n,\triangle,u_1,\cdots,u'_1,\cdots) \leftarrow relevant_p(n), \tilde{p}_{i_1}(n_1,u_1),\cdots,\tilde{p}_{j_1}(n'_1,u'_1),\cdots,$
            $par(n,n_1),\cdots,anc(n,n'_1),\cdots$
                where $(p,p_{i_x}) \in E_/$, $(p,p_{j_y}) \in E_{//}$
        $\delta \mathrel{+}:= \tilde{p}(n,n,\triangle,\cdots,\triangle) \leftarrow relevant_p(n), fun(n)$
        $\delta \mathrel{+}:= \tilde{p}(n,\triangle,\triangle,\cdots,\triangle) \leftarrow (p(n) = 1)$
end
```

Algorithm 3: Useful for Boolean queries

$q' \not\subseteq q$ 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[3] *$broughtBy(g, u)$.*

The complexity of Algorithm 3 is given by:

Theorem 7 *The problem of deciding, given an instance and a query, whether a function is useful, is* PTIME *in the size of I.*

6 Conclusion

To conclude, we mention some on-going implementation and related works.

The present work is motivated by the development of a P2P monitoring system based on flows of active documents in the style of ActiveXML [6].

Datalog has been intensively studied in the context of relational databases. Although efficient query optimization and incremental evaluation techniques for datalog have been proposed, such implementations are not easily available and would not be very suited to our XML context.

To validate our ideas, we have developed an algorithm tailored to our specific problem. The algorithm compiles a tree-pattern query q into an XQuery that essentially simulates the datalog programs of Sections 3 and 4. Let us for instance consider the example in Figure 3. The corresponding XQuery program is given in Figure 10.

Note that its output is one of the three values: 0, $\frac{1}{2}$ or 1. Observe also that it defines the variables $\$av, \$bv, \$cv$ (Lines 2, 10, 12, 28 and 30) that are the analogs of relations $\widehat{a}, \widehat{b}, \widehat{c}$ of the datalog program. Variable $\$sat$ holds the value of the *possible satisfiability* of the query rooted at a. As one can observe, the *possible satisfiability* of a tree-pattern query of root r is computed based on the *possible satisfiability* of the subqueries rooted at the children of r. Variables $\$sat1$ and $\$sat2$ hold the result of the computation for subqueries rooted at b (Lines 6-20) and c (Lines 24-39). Note also that if the subquery rooted at b has no chance to be satisfied, i.e. $\$sat1$ has a 0 value, the *possible satisfiability* of the subquery rooted at c is not computed (the condition of *if* in line 22 would be *true*) and the whole query evaluates to 0.

The XQuery that is obtained computes possible satisfiability, but not in an incremental manner. We are currently working on incremental evaluation techniques. Indeed, we are investigating two directions: (i) an incremental algorithm tailored to our particular problem, and (ii) a generic incremental algorithm for tree-pattern queries, somewhat the analog of incremental datalog evaluation in an XML setting.

Related Work

Our work on the maintenance of tree-pattern views over active XML documents relies primarily on previous works around datalog, e.g., [4, 11, 17, 16]. We have used this particular language to describe our algorithms so that we could benefit from both the optimization techniques QSQ [22] and Magic Set [10], and the incremental maintenance of datalog; see [4].

Connections between tree-pattern queries and monadic datalog have been studied in [14]. Tree-pattern queries over trees are closely related to XPATH expressions but they are differences in expressive power; see e.g., [18]. The problem of the incremental view maintenance for graph-shaped semistructured data is studied in [7].

Some recent works have considered the incremental maintenance of XPATH views over trees [20, 21]. The setting is different than ours, since data can be added and removed at any place in the document.Our model is more expressive since we can authorize the data to be appended at particular places in the document, i.e. the places where the *functions* reside. Remark the fact that we can simulate their setting by appending some function as child for each *data node* in the docu-

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

```
let $sat :=
 for $av in doc("test.xml")/a
 return
  let $initialSat := <value>1</value>
  return
   let $sat1 :=
   (
    let $valuesSon1 :=
    (
       ( for $bv in $av/b return <value>1</value> )
       union
       ( for $bv in $av/sc return <value>0.5</value> )
     )
     return
      <value>
      { if (count($valuesSon1) = 0) then 0
        else min((<value>{max($valuesSon1/node())}</value> union $initialSat)/node())
      }
      </value>
    )
    return
     if ($sat1 = 0) then <value>0</value>
     else
      let $sat2 :=
      (
       let $valuesSon2 :=
       (
         ( for $cv in $av/c return <value>1</value> )
         union
         ( for $cv in $av/sc return <value>0.5</value> )
      )
      return
      <value>
      {if (count($valuesSon2) = 0) then 0
       else
   min((<value>{max($valuesSon2/node())}</value> union <value>{$sat1}</value>)/node())
      }
      </value>
    )
    return $sat2
 return
  if (count($sat)=0) then 0
  else $sat/node()
```

Figure 10: XQuery program for the tree-pattern query example in Figure 3

ment. We also compute the *possible satisfiablity* of a query as well as the pertinent functions for this query. This way, if data is appended to the document by some useless functions, there is no need to process this newly appended data for maintaining the view.

Our complexity analysis for query evaluation on active documents has been influenced by works of [14, 15, 18]. In particular, the complexity for datalog evaluation on trees is studied in [14] whereas [15] considers the complexity of query evaluation for portions of XPATH; and [18], 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. A main aspect of their work is the detection of functions that may contribute to the answer so that should be activated. Although the techniques are very different because of the different context, our notion of useful function is in the same spirit.

Some complexity issues about active documents are studied in [8, 19]. Other works about active documents may be found at [9].

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 . Technical Report 331, INRIA Team Gemo, 2005.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] S. Abiteboul, I. Manolescu, and E. Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.

[6] S. Abiteboul and B. Marinoiu. Monitoring P2P systems. Technical report, INRIA Team Gemo, 2007.

[7] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[8] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *PODS*, 2001.

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

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

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

[12] 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.

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

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

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

[16] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques,

and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

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

[19] A. Muscholl, T. Schwentick, and L. Segoufin. Active context-free games. *Theory Comput. Syst.*, 39(1):237–276, 2006.

[20] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *WWW*, pages 671–681, 2005.

[21] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, pages 443–454, 2005.

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