Efficient Maintenance Techniques for Views over Active Documents*

Serge Abiteboul#, Pierre Bourhis#*, Bogdan Marinoiu#
#INRIA Saclay – Île-de-France and University Paris Sud *ENS Cachan

firstname.lastname@inria.fr

ABSTRACT

Many Web applications are based on dynamic interactions between Web components exchanging flows of information. Such a situation arises for instance in mashup systems or when monitoring distributed autonomous systems. Our work is in this challenging context that has generated recently a lot of attention; see Web 2.0. We introduce the axlog formal model for capturing such interactions and show how this model can be supported efficiently. The central component is the axlog widget defined by one tree-pattern query or more, over an active document (in the Active XML style) that includes some input streams of updates. A widget generates a stream of updates for each query, the updates that are needed to maintain the view corresponding to the query. We exploit an array of known technologies: datalog optimization techniques such as Differential or MagicSet, constraint query languages, and efficient XML filtering (YFilter). The novel optimization technique we propose is based on fundamental new notions: a relevance (different than that of MagicSet), satisfiability and provenance for active documents. We briefly discuss an implementation of an axlog engine, an application that we used to test the approach, and results of experiments.

1. INTRODUCTION

Many Web applications are based on dynamic interactions between Web components exchanging flows of information. Such a situation arises for instance in mashup systems [20] or when monitoring distributed autonomous systems [11]. The efficient management of flows of information is a challenging problem that has generated recently a lot of attention; see Web 2.0 [42]. Starting from datalog and Active XML technologies, we introduce a novel model for capturing interactions between Web components and show how it can be supported efficiently. We briefly present a system we implemented based on these ideas and discuss experiments.

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

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

The core of the model is a complex stream processor that we call *axlog widget*. The term axlog results from the marriage between Active XML (AXML for short) [5] and datalog [7]. An axlog widget consists of an active document interacting with the rest of the system via streams of updates. See Figure 1. The input streams specify updates to the document (in the spirit of RSS feeds). In most of the paper, the focus is on input streams where the updates are only insertions, although we do consider briefly streams with deletions. An output stream is defined by a query on the document. More precisely, it represents the list of update requests to maintain the view for the query. The queries we consider here are tree-pattern queries with value joins (and a template to produce an XML result). Our data model and queries include a time dimension, an essential feature for such a setting.

A set of widgets deployed on several peers and interacting by exchanging streams of updates forms an *axlog system*.

The main issue for widgets is the efficient computation of output streams, i.e. a view maintenance problem. A main contribution of the paper is a new algorithm for incrementally computing these output streams. The algorithm exploits known datalog optimization techniques such as Differential [15] and MagicSet [16]. Time is handled using previous works on constraint query languages. The core novelty of our algorithm is the use of a notion of relevance (of streams for a view) better adapted to a dynamic setting than that of MagicSet. This concept is based on two key notions for active documents: *provenance* of data and *satisfiability* of a fact. Provenance is in the spirit of [17]. Satisfiability allows stating whether some (incomplete) fact has a chance to hold in the future. Based on relevance, we show how to filter data *before* it enters the datalog program (to save in processing) and possibly at the source of the stream (to save on communication). We also see that, with this new notion of relevance, our algorithm is more optimistic (aggressive) than MagicSet.

A formal study of satisfiability (as well as relevance) can be found in a companion paper [6], with, in particular, results on the complexity of the satisfiability and relevance problems for a variety of languages and a number of contexts, e.g. functions with WSDL-style [43] signatures. Optimization is not considered there.

The work on axlog was motivated by the development of the P2PMonitor system, a system for monitoring P2P applications, introduced in [11]. The notion of axlog was not present in that paper, so nothing concerning axlog optimization. Beyond distributed monitoring, axlog widgets can be

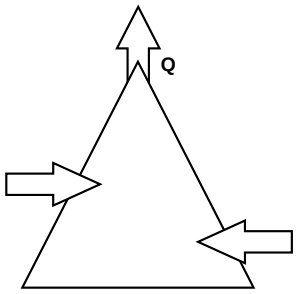

Figure 1: An axlog widget

used to support a number of tasks in a distributed environment such as organizing a choreography of Web services. We will briefly illustrate their use with a particular distributed application, the Dell supply chain [30].

Based on our maintenance optimization algorithm, we have implemented an axlog engine. This was demonstrated in [12] using the Dell supply chain application, together with the P2PMonitor system. We will briefly describe the implementation. To illustrate the gains achieved by our algorithm, we will present performance measures.

The paper is organized as follows. In Section 2, we formalize the notions of active document, query, update stream and axlog widget and briefly illustrate how axlog widgets can be used. In Section 3, we present our optimized algorithm for the maintenance of axlog widgets in the presence of insert-only streams. Deletions are considered in the next section. In Section 5, we discuss the implementation and some performance evaluation. Finally, we overview related works and conclude.

2. THE AXLOG MODEL

The main component in our approach is the *axlog widget*. We formalize this notion in this section as well as the streams they use as input/output. We next consider the frontiers of the system, i.e., how standard applications may produce or use streams and how a stream can be "published". Finally, we illustrate the use of widgets.

2.1 Axlog Widget

An axlog widget is mainly a complex stream processor, that is defined by one (Active)XML document and one or several queries. The widget receives update streams, by subscribing to Web services providing them, and generates output streams also as Web services. The content of each output stream is specified by a temporal query over the document. For instance, the axlog widget illustrated in Figure 1 has two update sources and generates one output stream that corresponds to the query Q. When the document evolves, the view defined by the query changes. The output stream consists of the sequence of updates to maintain the view. We next define formally the data model and the query language.

We assume the existence of some infinite alphabets: $\mathcal{I}$ of node identifiers, $\mathcal{L}$ of labels, $\mathcal{F}$ of function calls and $\mathcal{V}$ of variables. To simplify the presentation, we do not distinguish here between data, attributes and labels, i.e., our labels are meant to capture these three notions. (Our actual implementation does.) We use the symbols n, m, p for node identifiers, $a, b, c, ...$ for labels, $?f, ?g, ?h, ...$ for function calls,

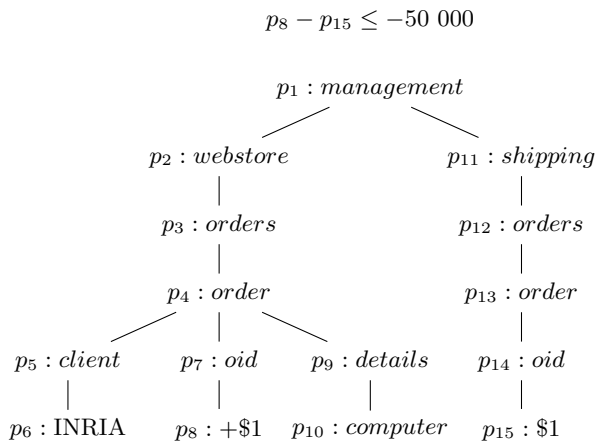

Figure 3: A tree-pattern query

and $\$1, \$2...$ for variables.

A function call $?f$ may be seen as a call to a Web service that brings a stream of messages, so as a call to a continuous service.

We consider active documents in the style of AXML [5, 14], ignoring here the ordering of siblings in the trees. To denote time, we use $\mathbf{Q}$, the set of rational numbers, a dense ordered domain. In general, such domains may also be used to capture other data such as space. To simplify the presentation, the only such domain we consider is time, that is essential for the kind of surveillance applications we are considering. The time we consider here is the time a particular node was introduced in the document since the creation of the document. Clearly, other notions of time may be considered. Formally, we have:

DEFINITION 1 (**Active Document**). An active document *is a triple* (t, λ, τ) *where (1)* t *is a finite binary relation that is a finite tree with* $nodes(t) \subset \mathcal{I}$ *; (2)* λ *is a labeling function over* $nodes(t)$ *with values in* $\mathcal{L} \cup \mathcal{F}$ *; (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* $\mathcal{F}$*) ; and (4) the* time-function τ *is a function from* $nodes(I)$ *to* $\mathbf{Q}$. A (data) forest *is a finite set of active documents, and of single-node trees labeled with symbols from* $\mathcal{F}$. *We also impose that in a document, the time of a node is larger or equal to that of its parent.*

We assume that a function call $?f$ does not occur twice in an active document. (We call $n_{?f}$ the node corresponding to a particular $?f$.)

An example of active document is given in Figure 2. Note that the root has time 0 by definition. In the representation, when a time value is not represented, we assume that the node inherits its time from its parent. So, in this document, the input streams have not brought any data yet.

To simplify the presentation, we consider here in a standard manner tree-pattern queries with joins. (One could consider more general queries, e.g., in XQuery.) An example of query is given in Figure 3. A pattern node is denoted $p_i : l_i$, where $p_i \in \mathcal{I}$ and $l_i \in \mathcal{L} \cup \mathcal{V}$. The single lines indicate a parent-child relationship, and the double lines an ancestor-descendant relationship. The dollar-variables and

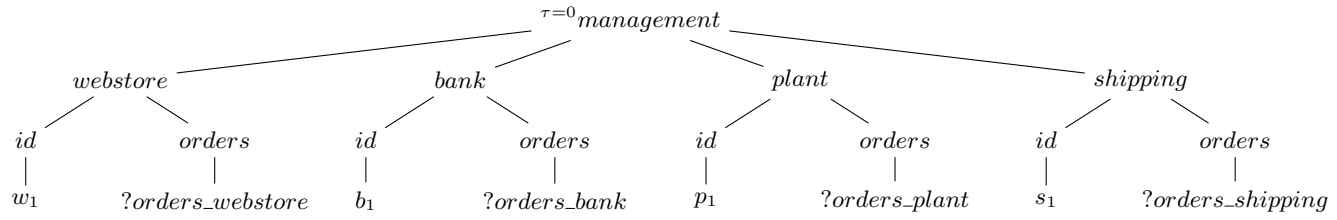

Figure 2: An active document

$*$ (the wildcard) may match any label. A label is requested to be in the result if marked by a "+". The time constraints[1] are of the form $\alpha_1 p_1 + \alpha_2 p_2 \leq \alpha$ where each p_i is a query node and the α's are in $\mathbf{Q}$. See, e.g., the constraint at the top of Figure 3.

DEFINITION 2 (**Tree-Pattern Time Query**). *A* query q *is an expression* $(E_/, E_{//}, \lambda, \mathcal{C}, \pi)$ *where:*

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

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

The semantics of queries is defined as follows.

DEFINITION 3. *Let* $q = (E_/, E_{//}, \lambda, \mathcal{C}, \pi)$ *be a query and* $I = (t', \lambda', \tau)$ *a document. A* valuation ν *from* q *to* (t', λ', τ) *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* $\lambda'(\nu(p)) = \lambda'(\nu(p'))$.
- *Time constraint-obeying : For each* $\alpha_1 p_1 + \alpha_2 p_2 \leq \alpha$ *in* $\mathcal{C}$*,* $\alpha_1 \tau(\nu(p_1)) + \alpha_2 \tau(\nu(p_2)) \leq \alpha$ *holds.*

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

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). Observe that by this definition, query results are tuples. We can obtain XML documents by restructuring this result using some template. (Details omitted.)

[1]One could in general introduce more complex time constraints in a post-processing phase. But the constraints considered here are those supported by our optimization technique and that often suffice in practice.

2.2 Update Streams

The functions in the documents receive streams of update requests. So, by extension, we speak of a stream $?f$ for the stream of update requests received by the function call $?f$. The elementary updates in a stream $?f$ are $ins(?f, t)$ and $del(?f, q)$ where t is an XML tree[2] and q is a Boolean tree-pattern query that returns the nodes to be deleted. (In practice, a delete query often uses identifiers to specify the trees to be deleted.) As result of the arrival of updates, an active document evolves in time. The *result* of applying $ins(?f, t)$ to a document I, denoted $ins(?f, t)(I)$, is the document obtained from I by adding a fresh copy of t as a sibling of the node labeled $?f$. The result of applying $del(?f, q)$ to a document I, denoted $del(?f, q)(I)$, is the document obtained by deleting the siblings of node $?f$ satisfying q as well as their descendants. Observe that both operations are in some sense local.

When applying an update $ins(?f, t)$ to a document I, we impose that (i) the time of each node in t is larger than the time of each node in I; and (ii) the times of all nodes in t are identical. Condition (i) is compulsory to be able to reason about time. Condition (ii) may be relaxed but is used here to simplify.

2.3 The Axlog Environment

We will describe an implementation of axlog widgets. In our implementation, the streams are implemented as *channels*. Channels are supported by a Pub/Sub mechanism based on Web Services. More precisely, a channel is exposed as a Web service to which axlog widgets may subscribe. (A list of subscribers is maintained for each channel by its owner). When subscribing to a channel, an axlog widget specifies a URI, i.e. the address of a Web Service that is called by the channel provider every time a new update becomes available for that channel.

The various widgets interact using these channels. The system of widgets also interacts with the rest of the world as follows:

- Input alerters: These are software components that observe a particular system and output source streams, in an axlog style. To illustrate, we mention two of the alerters we implemented. We implemented a Web server alerter that monitors the communications of the server. An XML notification is published on the channel each time particular in-call or out-call messages are detected. Also, we implemented an RSS alerter that

[2]In general, an insertion may bring an active document. To simplify, unless otherwise stated, we assume here that inserts only bring static trees.

regularly polls particular RSS feeds and sends XML notifications of changes when detected.

- Output publishers: Channels may be published outside the axlog system, e.g., using RSS feeds or emails, in databases or in files (e.g. as Web pages).

Note that a particular application may interact with axlog widgets using available alerters/publishers. It may also use the simple channel protocol to interact with them more directly.

2.4 Axlog at Work

Before delving in technicalities on axlog widgets, we illustrate their use. Consider the Dell supply chain application [30]. This distributed application represents the computer manufacturing platform of the Dell company. It involves customers, Web stores, plants for computer manufacturing, banks, suppliers, shipping companies and warehouses for the parts used by the plants. An order issued by some customer enters the system via the Web store. It arrives in a plant that obtains the relevant parts from a warehouse and assembles the product. Suppliers have to permanently supply this warehouse to avoid delays in obtaining the parts. After payment through a bank, the product is shipped.

In such an environment, the use of axlog widgets facilitates supporting tasks that are typically very complex because of the distribution. For instance, they turn to be very useful to gather statistics on the entire system, to detect bottlenecks and help optimize its usage. They also help, for instance, detecting parts reaching a dangerously low level in warehouses so that suppliers could ship new parts. Widgets may also be used to detect that the processing of some particular order took too much time, more than a certain threshold. For instance, this last task can be supported (simplifying somehow the setting for presentation) using a widget based on an active document and a query in the style of Figures 2 and 3 respectively.

The active document of Figure 2 uses alerters (and their channels) that are provided by the various partners, e.g., ?*order_webstore* is a function call to an alerter that detects the new orders and publishes them on a channel. Observe in the query, the use of *join* on the order identifiers and the time constraint. The query states that an interesting order is one that has been detected first at a purchasing site and then at a shipping company, with more than 50 000 (seconds) interval between the two events.

As mentioned in the introduction, the axlog principles and the optimization techniques for axlog were motivated by the development of the P2PMonitor system [11]. In [12], we have demonstrated the P2PMonitor system with the Dell supply chain application. The system that was presented already was axlog-based. It allowed high level subscriptions using active documents and tree-pattern queries and supported axlog widgets as stream operators.

3. VIEW MAINTENANCE: INSERTS ONLY

The core of our system consists of widgets that maintain views over active documents and publish the updates to a view on an output stream. In this section, we describe how to optimize this maintenance. This is achieved by combining a wide array of existing techniques on datalog-based query processing on trees, datalog optimization and stream filtering, and introducing novel features that are more specific to active documents.

In this section, we focus on insert-only streams and briefly consider streams with deletes in the next section.

3.1 Moving to datalog, Differential and MagicSet

To use datalog, we represent the document (a tree) using relations. We can express a query using datalog computations over trees in the style of [25]. To optimize the maintenance of this program, we use two known techniques for datalog, namely Differential for incremental computations [15] and MagicSet [16, 41] for query optimization. Since this is only combining known techniques, our presentation will be very brief.

We assume that the document is represented in a relational database using the extensional relations *root, child, descendant, label, function, time* with the standard meanings. In particular, $label(a, n)$ (respectively $function(n)$) holds if the node with identifier n is labeled by $a \in \mathcal{L}$ (respectively $?f \in \mathcal{F}$). A fact $time(t, n)$ holds if node n was introduced in the document at time t.

A tree-pattern query q is translated into a datalog program that computes the partial matchings to subtrees of the query bottom-up. Such a translation is easily obtained by recursion on the nodes of the tree-pattern query. See, e.g., [25]. Observe that the datalog programs we consider are in fact nonrecursive.

In practice, it is preferable to avoid constructing the *descendant* relation. That relation can be simulated using an identification scheme for tree nodes that makes it easy to check whether a node is a descendant of another [8]. We use an identification scheme such that no recomputation is needed when the tree evolves [36]. (Recall that we only consider inserts in this section, and deletes in next section, but no *move* update that would move a subtree from one place in the document to another.)

The datalog rule-based language is a well-studied query language for relational databases, see e.g. [7]. The rules in datalog are of the form:

$$r_0(x_1, \ldots, x_n) \leftarrow r_1(\overline{u_1}), \ldots, r_n(\overline{u_n}),$$

where each r_i is a relation name, each x_i a variable, each $\overline{u_i}$ a vector of variables or constants, $r_0(x_1, \ldots, x_n)$ is called the *head* of the rule, while $r_1(\overline{u_1}), \ldots, r_n(\overline{u_n})$ form the *body*. We also impose that each variable occurring in the head also occurs in the body. A result of the application of the rule is a fact $r_0(\nu(x_1), \ldots, \nu(x_n))$ for some valuation ν of the variables occurring in the rules, such that $\nu(body)$ holds.

To each node r of q corresponds a relation denoted R_r. As an example, datalog Program 1 computes the results of the query in Figure 3.

We want to avoid deriving irrelevant facts. To do that, we use the MagicSet technique, that rewrites the datalog program (given the view query) into one that derives only facts that are "relevant" for the view. Also, we want to avoid recomputing the view when new updates arrive. For that, we use the Differential technique for the incremental maintenance of datalog views. This technique prevents us from unnecessarily repeating the same datalog derivations when a stream brings new data. The combination of these various techniques is already the source of important savings.

We next illustrate by example that we can do much better.

Datalog Program 1: for Query q of Figure 3

begin

$q(x) \leftarrow R_{p_1}(n,x)$

$R_{p_1}(n,x) \leftarrow root(n), label(management,n),$
$child(n,n'), R_{p_2}(n',x,t'), child(n,n''),$
$R_{p_{11}}(n'',x,t''), t'' - t' \geq 50000$

$R_{p_2}(n,x,t) \leftarrow label(webstore,n), child(n,n'),$
$R_{p_3}(n',x,t)$

$R_{p_3}(n,x,t) \leftarrow label(orders,n), child(n,n'),$
$R_{p_4}(n',x,t)$

$R_{p_4}(n,x,t) \leftarrow label(order,n), child(n,n'), R_{p_5}(n'),$
$child(n,n''), R_{p_7}(n'',x,t),$
$child(n,n'''), R_{p_9}(n''')$

$R_{p_5}(n) \leftarrow label(client,n), child(n,n'), R_{p_6}(n')$

$R_{p_6}(n) \leftarrow label(INRIA,n)$

$R_{p_7}(n,x,t) \leftarrow label(oid,n), child(n,n'), R_{p_8}(n',x,t)$

$R_{p_8}(n,x,t) \leftarrow label(x,n), time(t,n)$

$R_{p_9}(n) \leftarrow label(details,n), child(n,n'), R_{p_{10}}(n')$

$R_{p_{10}}(n) \leftarrow label(computer,n)$

$R_{p_{11}}(n,x,t) \leftarrow label(shipping,n), child(n,n'),$
$R_{p_{12}}(n',x,t)$

$R_{p_{12}}(n,x,t) \leftarrow label(orders,n), child(n,n'),$
$R_{p_{13}}(n',x,t)$

$R_{p_{13}}(n,x,t) \leftarrow label(order,n), child(n,n'),$
$R_{p_{14}}(n',x,t)$

$R_{p_{14}}(n,x,t) \leftarrow label(oid,n), child(n,n'), R_{p_{15}}(n',x,t)$

$R_{p_{15}}(n,x,t) \leftarrow label(x,n), time(t,n)$

end

Consider Figure 4. Ignore q_2 for now. So consider Query q_1 and the document I. Observe that the only relevant data that $?f$ may provide for the query is a (tree with root) y. So we would not change the view state if we filter $?f$ to only keep this y, if produced. This is first possibly providing important saving in processing because we introduce less data in the datalog program, so we save on useless tests and derivations. Also, this may result in important saving in communications if the source of stream $?f$ is remote. Furthermore, if $?f$ produces y, the stream cannot contribute to the view anymore and the corresponding subscription may be discarded. Suppose on the contrary that the function call $?f$ does not produce y and terminates, i.e. sends an End-of-Stream message. Then whatever $?g$ brings, the view will remain empty and it would be a waste of effort to keep testing for the left subtree of q_1. More generally, what we need is an analysis of the (dynamic) situation that detects the streams that are useless for some view, and more precisely, the kinds of data that is expected from each particular stream so that we can filter out irrelevant data. Based on the previous discussion, it should be clear how such relevance information could be used (i) to unsubscribe to streams, (ii) reduce the quantity of data that is introduced in our program and (iii) (if the filter is installed remotely) reduce communications. It should also be clear that the relevance of streams evolves in time.

Towards this goal, we show how to evaluate a novel notion of relevance much better adapted to our context than the classic relevance of MagicSet. To do that, we also consider two notions that are interesting in their own right, *satisfiability* of facts and *provenance* of data. The second one is in the spirit of notions considered in, e.g., [17]. We then show how the concept of relevance is used to optimize the computation by filtering the input streams. So, the general algorithm works as follows. (See Figure 5.) In a "tuning"

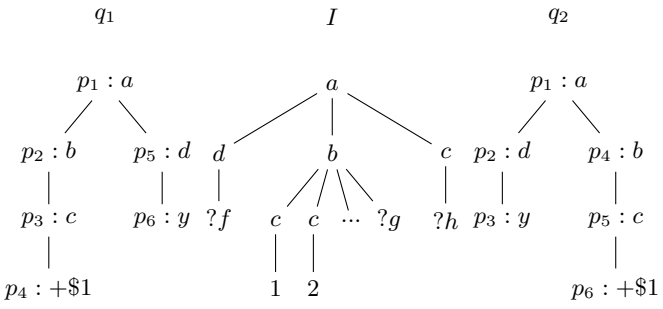

Figure 4: Beyond MagicSet

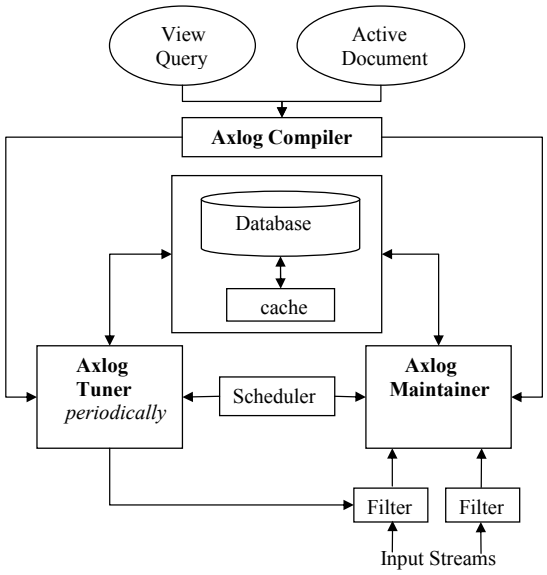

Figure 5: Architecture of the axlog engine

phase, we compute satisfiability, provenance and relevance information. Based on that, we install some filters on the streams. We then maintain incrementally the view. Periodically, we perform a new tuning phase. We next describe in turn the computation of these three notions.

3.2 Satisfiability

We say that a tuple u is *satisfiable* for a query q and an active document d if there exists a (possibly empty) sequence of stream updates ω such that $u \in q(\omega(d))$. A query is satisfiable for an active document if there is at least one satisfiable tuple for that query and that active document. Observe that this notion is interesting in its own right. For instance, one may want to ask, in an on-going soccer tournament, whether Lyon and Liverpool still have a chance to play against each other. Note that the combination of satisfaction and satisfiability leads to some form of 3-valued logic, where a tuple may be true, false for now but true in some possible future, or false forever.

A main difficulty with satisfiability is that the set of sat-

Datalog Program 2: for Query q_2 of Figure 4

begin

$q(x) \leftarrow R_{p_1}(n, x)$

$R_{p_1}(n, x) \leftarrow root(n), label(a, n), child(n, n'),$
$child(n, n''), R_{p_2}(n'), R_{p_4}(n'', x)$

$R_{p_2}(n) \leftarrow child(n, n'), label(d, n), R_{p_3}(n')$

$R_{p_3}(n) \leftarrow label(y, n)$

$R_{p_4}(n, x) \leftarrow label(b, n), child(n, n'), R_{p_5}(n', x)$

$R_{p_5}(n, x) \leftarrow label(c, n), child(n, n'), R_{p_6}(n', x)$

$R_{p_6}(n, x) \leftarrow label(x, n)$

end

isfiable tuples for a query q is possibly infinite. We use a finite representation for this set based on "generalized tuples" [29]. A *generalized tuple* is a tuple with values in a set of variables that are constrained by a system of equality and inequality constraints. Here we are also concerned with comparisons of the form $\alpha_i \$_i + \alpha_j \$_j \leq \alpha$ because of the time constraints. (Note that a "standard" tuple is a particular case of the generalized tuple but that a generalized tuple in general represents an infinite set of tuples.) We extend datalog to generalized tuples using unification.

Datalog Program 3: Optimized Program for computing satisfiability for Query q_2 of Figure 4

begin

$q(x, sat) \leftarrow R_{p_1}(n, x, sat)$

$\widehat{R}_{p_1}(n) \leftarrow root(n), label(a, n)$

$\widehat{R}_{p_2}(n) \leftarrow \widehat{R}_{p_1}(n'), child(n', n), label(d, n)$

$R_{p_3}(n, 1) \leftarrow \widehat{R}_{p_2}(n'), child(n', n), label(y, n)$

$R_{p_3}(n, \frac{1}{2}) \leftarrow \widehat{R}_{p_2}(n'), child(n', n), function(n)$

$R_{p_2}(n, sat) \leftarrow R_{p_3}(n', sat), child(n, n')$

$R_{p_2}(n, \frac{1}{2}) \leftarrow \widehat{R}_{p_1}(n'), child(n', n), function(n)$

$\widehat{R}_{p_4}(n) \leftarrow \widehat{R}_{p_1}(n'), child(n', n''), R_{p_2}(n'', sat),$
$child(n', n), label(b, n)$

$\widehat{R}_{p_5}(n) \leftarrow \widehat{R}_{p_4}(n'), child(n', n), label(c, n)$

$R_{p_6}(n, x, 1) \leftarrow \widehat{R}_{p_5}(n'), child(n', n), label(x, n)$

$R_{p_6}(n, \$1, \frac{1}{2}) \leftarrow \widehat{R}_{p_5}(n'), child(n', n), function(n)$

$R_{p_5}(n, x, sat) \leftarrow R_{p_6}(n', x, sat), child(n, n')$

$R_{p_5}(n, \$1, \frac{1}{2}) \leftarrow \widehat{R}_{p_4}(n'), child(n', n), function(n)$

$R_{p_4}(n, x, sat) \leftarrow R_{p_5}(n', x, sat), child(n, n')$

$R_{p_4}(n, \$1, \frac{1}{2}) \leftarrow \widehat{R}_{p_1}(n'), child(n', n''),$
$R_{p_2}(n'', sat), child(n', n), function(n)$

$R_{p_1}(n, x, sat) \leftarrow child(n, n'), R_{p_2}(n', sat_1),$
$child(n, n''), R_{p_4}(n'', x, sat_2), Min_2(sat_1, sat_2, sat)$

end

For a generalized tuple u and a query q, we say that u is a *satisfiable* tuple for q and I, if for each *instantiation* θ of the variables, the (standard) tuple $\theta(u)$ is satisfiable for q and I.

As mentioned in the introduction, a formal study of the satisfiability (and of the relevance) can be found in a companion paper [6], with, in particular, results on the complexity of the satisfiability and relevance problem for a variety of languages and a number of contexts (e.g. functions with WSDL-style [43] signatures). In the present paper, we are concerned with optimization.

We modify the datalog program to compute both the satisfied (complete) and satisfiable (incomplete) tuples. We also use MagicSet for computing satisfiability. Observe that the complete facts derived by this program would actually be derived by the MagicSet computation after the streams inserted data into the document. (Also, the incomplete facts correspond to sets of facts that would be derived). So, there are typically more facts inferred for the computation of satisfiability than for that of satisfaction.

As an example, consider the datalog Program 2. It computes the results of the query q_2 in Figure 4. The datalog Program 3 computes both satisfaction and satisfiability of facts for that query. (It includes the MagicSet optimization but not Differential, that is not relevant to this discussion.) To each node r of q_2 correspond now two relations denoted R_r and $\widehat{R}_r$. The computation is done in two phases. The *top-down* phase resembles to the classic program, but involves the $\widehat{R}_r$ relations in the head of the rules. The *bottom-up* phase takes into account the fact that inserts may result in adding subtrees as sibling to functions, and that those subtrees may match subqueries of q_2. That is why, for each relation R_r, two rules that have R_r as the head are added for the bottom-up phase: one that propagates tuples from the bottom, and the other makes up for tuples "created" by functions. Remark also the form of an atom $R_r(n, x_1, ..., x_k, sat)$, where n stands for the node identifier, $x_1, ..., x_k$ stand for the variables in the subtree rooted at r and the value of sat is $\frac{1}{2}$ if it is satisfiable, and 1 if the tuple is satisfiable *and* satisfied. A function call node introduces only satisfiable tuples. When deriving facts for a pattern node, one has to choose the satisfiability value for the fact as the minimum between the satisfiability values of the facts for the children of that pattern node. This is the role of the Min_2 atom in the program. $Min_2(x, y, x)$ holds if $x \leq y$, otherwise $Min_2(x, y, y)$ holds. In general, Min_k is present. This is defined in terms of Min_{k-1} and Min_2. (Details are omitted.) Remark also the presence of R_{p_2} atoms in the rules that have $\widehat{R}_{p_4}$ in the head. This is a feature inherited from Magic Set, which rules that the evaluation of the left branch (query rooted at p_2) has an impact on the evaluation of the right branch (query rooted at p_4). Thus derivations of useless facts are avoided.

Note that satisfiability is not a monotone feature, notably because of time constraints. To see that, suppose that at time 0, we have a document that consists of root r that has only one child, a function call $?f$. Suppose also that we are interested in the query q_{100} that states that r has an a child of time t with $t \leq 100$. Then the query is satisfiable. Now suppose that $?f$ receives some tree rooted with a b at time 123. Then we know that no matter which data will arrive, it will have a timestamp of more than 123. Thus q_{100} became unsatisfiable. Formally, the arrival of the b-rooted subtree did not only introduce data but also some new constraint: everything that will be brought by $?f$ will have a timestamp greater than 123. As we will see, the relevance of function call nodes is also nonmonotone. This is why we separate the tuning phase, that is evaluated periodically, from the processing phase, that runs continuously and maintains the view incrementally.

One could consider maintaining these notions incrementally as we do for satisfaction, using some Differential-like technique. However, such computations are typically quite expensive and it is unclear whether the gain would compensate the overhead. This is left for future research. We consider here that they are recomputed periodically, possibly in parallel with the satisfaction computation so that it is not necessary to block processing.

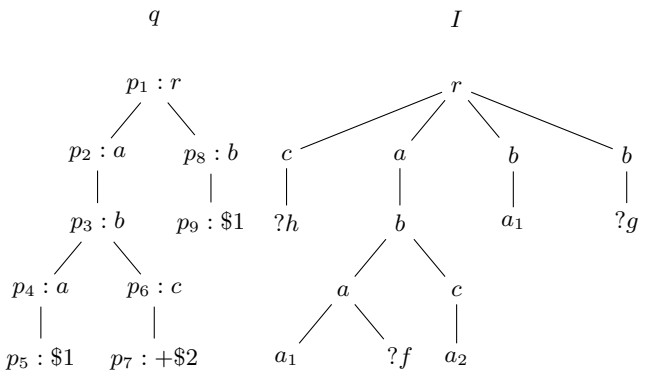

Figure 6: Example for provenance and relevance

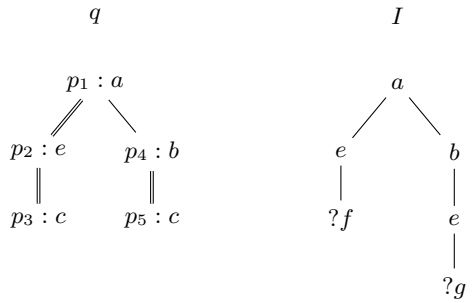

Figure 7: Example for hard relevance

3.3 Provenance

We reconsider the notion of generalized tuple to also include some "provenance" information. Generally, provenance is used to capture where data *came from*. Here, we are concerned with where data *might come from*. A generalized fact is now of the form $r(u_1, ..., u_n, \mathcal{C}, \mathcal{P})$ for some n-ary relation r, where each u_i is some constant or a variable, $\mathcal{C}$ is the set of constraints, and $\mathcal{P}$ is the provenance information defined as follows.

To simplify the presentation we ignore the time constraints here.

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

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

Observe that $u_1, ..., u_n$ may have no variable, but the fact still be unsatisfied because its truth is conditional to the arrival on some stream of some data that matches certain patterns.

We modify the datalog program that computes satisfiability so that it also computes provenance information. For the example of Figure 6, four tuples are derived:
$(a_2, \star, \star, \star, \star, \star, \star, \star, \star, \star)$, $(a_2, \star, \star, \star, \star, \star, \star, \star, \star, n_{?g})$,
$(a_2, \star, \star, \star, \star, n_{?f}, \star, \star, \star, \star)$, $(a_2, \star, \star, \star, \star, n_{?f}, \star, \star, \star, n_{?g})$,
where the first entry of the each tuple corresponds to the output variable \$2, and the last 9 entries, to the provenance for the 9 query nodes. The data components of the tuples that are derived are identical, i.e., a_2, but they have distinct provenance. By observing these tuples, one may be led to believe that $?f$ may bring useful data. But since we already obtained a_2, it turns out that this is not the case. This motivates the following notion.

Let $t = (u_1, ..., u_m)$, $t' = (v_1, ..., v_m)$ be two tuples where the first n entries are data and the last $n-m$ are provenance. We say that t is *dominated* by t', denoted $t \prec t'$ if there exists a renaming $t'' = (w_1, ..., w_m)$ of t such that: (a) for each $1 \leq i \leq n$, $v_i = w_i$, and (b) for each $n+1 \leq i \leq m$, either $v_i = \star$ or $v_i = w_i$. The intuition is that any relevant data needed by the dominating tuple to lead to satisfied tuples is also needed by the dominated one. Thus, the dominated tuples are useless because they lead to the same satisfied tuples, but they need more data than the dominating ones.

We refine the set of candidates by eliminating the dominated tuples.

In the example, the last three tuples are eliminated because the first tuple contains only document-provenance, so the tuple is already satisfied by the current instance of the document.

3.4 Axlog-relevance

We are now ready to describe the computation of axlog-relevance.

We compute satisfiability with the provenance information. We eliminate from the result of the satisfiability computation the dominated tuples; observe that this step is not monotone (i.e., requires using relational calculus). Finally, we compute axlog-relevance as follows. A function call $?f$ is *axlog-relevant* for p_i if $n_{?f}$ (the node corresponding to $?f$) occurs in the column corresponding to p_i in the provenance $\mathcal{P}$ of some remaining tuple. Based on this, the unary relations $relevant[p_i]$ are computed. The meaning of $relevant[p_i](n_{?f})$ for some function call node $n_{?f}$ in the document is that, $?f$ is *axlog-relevant* for p_i. This concludes our presentation of the computation of axlog relevance.

Axlog relevance provides a sufficient criteria in the sense that one can filter the stream provided by some $?f$ with the union of the patterns p_i such that $n_{?f}$ is axlog-relevant for p_i, without affecting the view. One may question whether each of these p_i is also "necessary", i.e., whether one could use a tighter filter by removing some p_i's from the disjunction and still not affect the value of the view. We briefly show that this is not the case.

Remark: The example in Figure 7 illustrates the subtlety of the problem. Our algorithm would compute that $n_{?f}$ is axlog-relevant for p_3. Indeed, intuitively this seems to be the case since if $?f$ brings a c, the p_2 branch of the query

is matched. Note, however, that we also have to match the second branch, which may be achieved only with $?g$ returning a subtree t with a label c in it. But t would also match the first branch of the query as well, so the data returned by $?f$ is not necessary. It is shown in [6] that some semantic notion of relevance may be defined and tested, but that this requires in particular testing tree-pattern query containment [33] and cannot be realized in relational calculus.

To conclude this discussion of relevance and before turning to using that notion for filtering the input streams, observe that one can in a very similar manner define a notion of *data relevance*. More precisely, a node (and its entire subtree) is not relevant if it can be removed without altering the value of the view. This could be the basis for useful garbage collection, but will not be considered here.

3.5 Filtering the Streams

We are now going to use the $relevant[p_i]$ relations to install filters on input streams that filter out useless data, before entering the datalog program. The previous analysis will guarantee that this filtering has no chance to impact the view, in any future. The filtering can be processed efficiently using stream filtering techniques.

Let $\{p_j\}$ be the set of query nodes in q, such that $n_{?f}$ is relevant for each p_j. As already mentioned, we can filter the stream $?f$ with $\vee p_j$ with no effect on the state of the view. In particular, if $n_{?f}$ is not relevant for any p_j, we can delete entirely this stream, i.e., the widget can unsubscribe to $?f$. If $n_{?f}$ is relevant for some, we evaluate directly the tree-pattern queries $\vee p_j$ on the data trees of the input streams (before transforming them into relational tuples). For this, we use a filter based on YFilter [19]. The YFilter does not handle joins, so we process the nonjoin part of the filter and then filter out by the join conditions in a second phase. XML trees are fed in the filter that output tuples that correspond to matchings of some p_i. (Observe that the arity of the tuple depends on the p_i that matched.)

We will detail the processing involving these filters. Observe that in the original (pure datalog) spirit of the technique, a new data tree arriving in an input stream is translated into relational tuples that are inserted as a new Δ into the relational image of the active documents. Instead, for each p_i such that $n_{?f}$ is relevant for p_i, we can filter the XML stream $?f$ with p_i. The filter produces fewer facts that are fed directly in the datalog program.

To illustrate the use of filters, consider the query q_2 and the document I in Figure 4, where the p_is represent node identifiers of the query. Observe that before we can derive some result, the tree-pattern rooted at p_5 has to be matched to data returned by $?g$. This leads to installing a p_5 filter on the stream $?g$. This filter produces binary tuples in a relation $filter[p_5]$, where $filter[p_5](\$fun, \$1)$ means that some tree in the stream $\$fun$ matched query p_5 and produced $\$1$ as result. The algorithm previously discussed will compute a unary relation, namely $relevant[p_5](\$fun)$, that will contain the identifiers of relevant function nodes for p_5. In our case, $relevant[p_5] = \{(n_{?g})\}$. Then the interface between the filter and the datalog program consists in rules of the form:

$$R_{p_5}(\$fun, \$1) \leftarrow relevant[p_5](\$fun), filter[p_5](\$fun, \$1)$$

There will be one such rule for each p_i. For the considered example, these rules will be added to those of Program 2 that computes satisfaction.

When possible, the filtering is performed at the source of the input stream which results in large communication savings. A lot of saving can also be achieved by performing these filtering efficiently, and in particular, by combining several filters on the same stream into one unique filter. This is even of greater importance if we consider that one stream may be shared as input by many axlog widgets with different needs. XML stream filtering are generally based on automata, either non-deterministic, [19], or deterministic [26]. We use YFilter [19], one of the NFA-based solutions, because it scales very well with the number of queries on a stream.

3.6 A More Optimistic Strategy

Recall that in the tuning phase, we first computed satisfiability. Once we have computed the relevance of function calls and the filters, one could ignore all the facts derived for satisfiability and run from scratch a computation of satisfaction (with MagicSet and Differential). We choose instead to keep the facts that have been inferred. (In the implementation, both tuning and view maintenance run in the same database).

It turns out that this has a very interesting effect. The crux of MagicSet is to "focus on relevant data" (for a given query). However, some data may seem irrelevant for now but may be relevant assuming some data is received in the future. Satisfiability computes "data possibly relevant in the future". This results in a much more optimistic strategy than if we were simply to use MagicSet. When pure MagicSet blocks the computation of relevant facts because some not yet true fact is encountered, our technique is more tolerant and continues if this fact has a chance to become true in the future. As a consequence, the algorithm is likely to react faster to the arrival of new facts. On the other hand, it may be deriving facts that will never be relevant in the MagicSet sense.

To illustrate, consider again Figure 4. Observe that, q_1 and q_2 are the same query. However, the MagicSet evaluation depends on a choice of an ordering of the query branches (e.g. left is considered before right). Indeed, the notion of relevance as used in MagicSet depends heavily on this ordering. In the figure, the distinction between q_1 and q_2 illustrates these alternative evaluations of the same query. Now consider again q_2 of Figure 4. When evaluating q_2, the y branch is tested before the other branch. Suppose the data consists of a large collection of subtrees having roots labeled c (brought by $?g$). Observe that until $?f$ produces a node labeled y (for yes), the c subtrees do not produce any answer. If we evaluate the query q_2 with MagicSet, no tuple is produced until the y is received. In particular, the b subtrees are not even tested. Then, when y arrives, we have to perform a lot of computation and our response time will be bad. Since we know that the f function call *may* return a node labeled y, it seems more appropriate to be optimistic and start testing the b subtrees in advance. If we keep the facts derived for computing satisfiability, we more optimistically test the b-subtrees. Of course, a sophisticated optimizer may choose to use this optimistic strategy or not based on the application. In the prototype, we chose the option to use the tuples derived for computing satisfiability when computing satisfaction.

We conclude this section with a remark on distributed

query optimization.

Remark: We have shown in this paper how to optimize an axlog widget individually. One may want to split such a widget in several components and distribute the computation on several machines. Also, one may want to globally optimize an axlog system consisting of several axlog widgets interacting via update streams. This brings in a number of new issues and opportunities for optimization. First, one can articulate the problem as the computation of a distributed datalog program [35]. We can again use techniques such as MagicSet to optimize it globally in the style of *dQSQ* [3]. One could also consider using a distributed query optimizer such as Optimax [9, 10].

4. MORE NONMONOTONIC FEATURES

We first briefly discuss deletions. Recall that a delete message may delete sibling trees of the function that receives such a message, as explained in Section 2. This may lead to invalidating facts that have been previously inferred. To evaluate the view, we can use incremental techniques to maintain views in presence of deletions [28]. In particular, we can use a Counting algorithm that rewrites the original datalog program in a similar style as Differential. The algorithm maintains counters of the numbers of alternative derivations for each derived fact.

The techniques that we developed for insert-only streams do not transfer easily to stream with deletes. In particular, the introduction of filtering is much more complex (because relevance is more complex). In particular, some data is relevant if it can match the condition of a delete message in some stream. In the document I of Figure 8, $?g$ is a delete-only stream, and $?f$ is an insert-only stream. One could think that a filter p_4 could be installed on $?f$ without affecting the view. Unfortunately, this would be wrong; think for instance of a message $del(?g, a/b/c/d/e)$.

Now consider queries with negation. Observe first that even if the input streams are insert-only, the output stream may need deletions, these are some belief revisions based on new facts that arrived. It is again possible to transform Differential so that it handles negation in queries. Now consider optimization. Some theoretical results on satisfiability in this context can be found in [6]. It is shown there that satisfiability is undecidable in general. This is an indication that an approach based on relevance is likely to be very complex.

We conclude this section with a very useful nonmonotone feature, end-of-stream, that may be seen in some way as deleting a function call node. An End-of-Stream message indicates that a stream has been closed. This is clearly a nonmonotone feature since some satisfiable tuples may become unsatisfiable as a consequence of an EoS message. There is no particular difficulty in handling them. They are simply taken into consideration during the tuning phase.

5. IMPLEMENTATION

To support axlog widgets, we implemented in Java an axlog engine that consists of three modules: a compiler, a tuner and a maintainer (see Figure 5). The system only supports inserts. We use the AXML System V2 for the management of active documents, i.e. for storing them, querying/updating them, and activating service calls in them. In

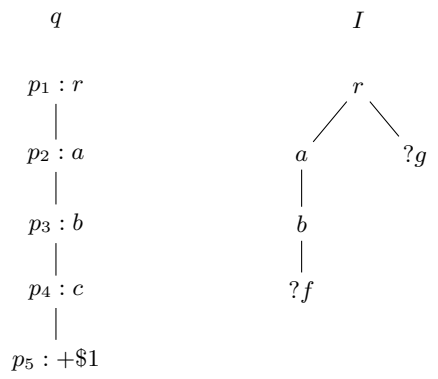

Figure 8: Example for nonmonotonic features

AXML System V2, storage persistence is supported using *eXist* [21] (a native XML database). To simplify the implementation, we also use *eXist* as storage for the axlog maintainer and tuner. Observe that, since the data is relational, one could have used a relational engine for that part of the system. We next briefly discuss the three modules.

5.1 Compilation

A new widget is specified using a query and an active document that may be given explicitly or as an URI (possibly inside an AXML store). When it receives it, the compiler constructs i) a datalog program that computes satisfaction for this query and this document, ii) a datalog program that computes satisfiability and provenance, iii) an *initialization* XQuery query. The initialization query is evaluated on the document and the extensional relations of the datalog programs are initialized.

5.2 Tuning

The tuner activates the datalog programs to compute satisfiability and provenance. The relevance of streams is evaluated. Based on that, some streams are ignored and filters are built for others and initialized. For filtering, we use the YFilter algorithm of [19]. We introduced a post-processing filtering phase, e.g. for joins in tree-patterns.

The tuning phase takes place first immediately after compilation, and then periodically (under the control of a scheduler), e.g. every hour or every day, as long as the axlog widget is active. During the first tuning phase, the processing is more important than in the successive tuning phases because lots of facts need to be derived by the datalog program that computes satisfiability. Observe that during subsequent phases, some satisfiability facts may be invalidated because of time constraints.

5.3 View Maintenance

At the beginning of this phase, only the datalog program for computing satisfaction and the filters are activated. At runtime, new items arrive on the input streams. Note that the current maintainer does not support active inputs (on the stream), that would require some form of compilation at runtime. (More precisely, they are ignored until the next tuning phase that may decide to complement the compi-

lation for the new function calls.) The input streams are fed into the filters. The output of the filters is transformed into relational data and is fed directly in the relations of the datalog program. New facts may be derived and new XML results produced. A main issue for the maintainer is the management of data. In some applications, messages in the streams are very rare, so we want the datalog data to reside on disk. In others, there is a continuous flow of messages and we want the data to stay in memory. So we implemented two very simple strategies that we call Cache and NoCache.

The NoCache strategy works as follows. To process the arrival of a new update (or a bunch of updates), the axlog maintainer brings data from its database. To avoid reloading the same data several times, it keeps a data structure that indicates which data have already been loaded (more precisely, the conditions that meet the data that have already been loaded in memory.) This data structure is updated each time the maintainer sends a query to the database. When the maintainer completes the processing of the current update (or bunch of updates), it updates the database (with the newly derived tuples) and the memory space is freed.

The Cache strategy works similarly except that the space is not reclaimed at the end of one update processing phase. When a certain memory threshold is reached only, some memory space is freed.

Clearly, one can use more complex caching strategies.

We included these two strategies in our experiments to measure the not-surprising impact of caching in this context.

5.4 Experiments

All the experiments presented in this paper have been performed on a computer with a 1.86GHz Pentium M processor and 1 GB of RAM memory. To simplify, we consider for update here, the insertion of a single tree. Batch insertions (i.e., several insertions of trees simultaneously) bring important savings compared to one tree at a time. Since this is a rather standard technique, we consider here only insertions of one tree at a time.

In the experiments, we wanted first to show the gain resulting only from using the concept of relevance and filtering of input streams. We use for baseline an algorithm called *Reeval*, that computes naively the difference between the views before and after update. The *Filter* algorithm (that is mentioned in Figures 9 and 10) uses only filtering of the input streams. The *Axlog* algorithm uses the array of techniques described in this paper. We consider two versions, with and without cache.

We have implemented a generator of tree-pattern queries and of active documents. (We verify that the query is satisfiable for the document.) The generator takes as input a number of parameters: e.g. the size of the documents (number of elements), the size of tree-pattern queries, the number of //-links, the number of function nodes. A test consists of a tree-pattern query, an active document and a number of updates for the function calls present in the document.

In the streams of updates produced by the generator, we take into account another parameter, namely the *pertinence ratio*. A *pertinence ratio* of p means that one update has p probability to bring to the document data that is relevant with respect to the tree-pattern query, so to have an impact on the view maintenance.

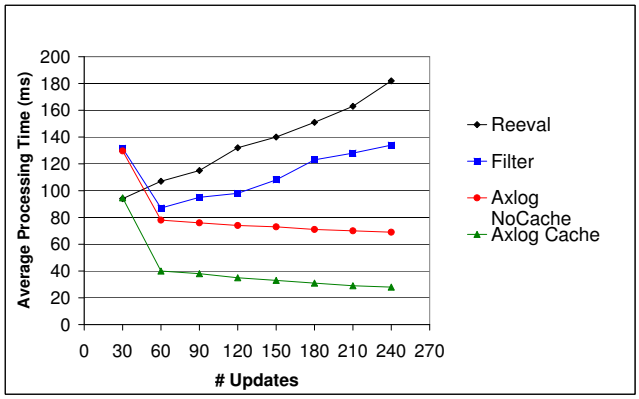

Figure 9: Average Processing Time dependence on the number of updates

For the experiments, we report on runs of 10 tests, each with a randomly generated widget. The measures we give are averaged on the 10 tests. The tree-pattern queries have 10 nodes, 2 //-links and more than 4 paths each. They also have at least one join condition . Each update inserts a small tree (about 20 nodes). The initial size of the document is fixed to 5 000 nodes.

We measure the different algorithms when the number of updates varies and when the data brought by streams is more or less relevant.

We evaluate the time to compute the output for each new update. This is a good indicator both for the system's reactivity and for the overhead brought by the view maintenance algorithms. Our analysis takes also into account the time spent by the algorithm in compilation and tuning. We focus on the first (compilation, tuning, maintenance) round. We count as processing time for an update the time the entire round took divided by the number of updates that are considered in this particular round. Note that this is pessimistic, since in subsequent rounds, there will be no compilation and tuning will typically be less expensive.

In the first experiment, the document includes 10 function nodes (streams), the pertinence factor is 70% (we expect that 7 updates pass the filters for every 10 generated) and we vary the number of updates from 10 to 250. At the end of the experiment, the size of the document has almost doubled. We measure the average update processing for each of the aforementioned algorithms. See Figure 9. One observes that Filter already brings some saving. Axlog brings more saving. Its saving increases with time by the effect primarily of Differential. Finally caching brings even more saving.

In a second experiment, we study how response time depends on the pertinence of the incoming data. As in the previous experiment, the document had an initial size of 5 000 nodes. The number of updates is fixed to 100. See Figure 10. Not surprisingly, the response time increases with the pertinence factor because more data is relevant, so enters the datalog program. Observe that the Axlog algorithms outperform Filter, because of Differential again. The gain increases with pertinence. For large pertinence, Axlog Cache largely outperforms all others.

In general, the gains brought by our technique are very substantial. It is particularly so, for streams that run for a

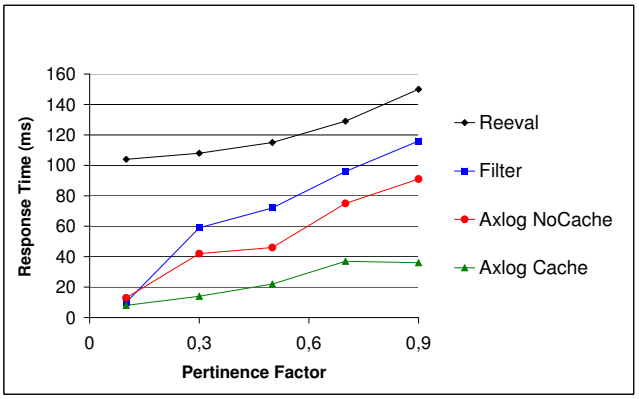

Figure 10: Average Response Time dependence on update pertinence factor

long time.

The improvements from our technique come at a price, that of some extra compilation and that of tuning. Let us call "initialization" the combination of compilation and the first tuning phase. In the experiments that we presented here, initialization took about 1.5 seconds. Amortized over the different updates, it is typically marginal after a relatively small number of updates. The time spent in it depends on the size of the document but also on the performance of the XQuery processor. For example, with *eXist*, initialization for a reasonably complex query (that previously used) and a 20 000 nodes document takes about 10 seconds.

The tests reported here have been conducted with documents of about 5 000 nodes and 10 function nodes. Similar patterns were observed in many tests.

For instance, we have also performed tests on smaller documents with good performance gains. If we consider the tests for documents with 100 nodes, the time for the complete reevaluation of the queries decreases 33%, but the planning time compensates because it decreases also about 60%. The gains will roughly be the same, especially after the updates bring significant data into the document.

We also verified that the technique scales to documents with a (reasonably) large number of function calls. We found that the response time does not depend much on the number of function calls. On the other hand, initialization does but in a reasonable way. For example, for a medium size document (5 000 nodes) with 180 function calls and queries as used in the previous tests, initialization took about 5 seconds.

6. CONCLUSION

Our work is based on previous works on incremental view maintenance [15, 18, 27, 28]. We use datalog and benefit from known techniques, Magic Set [16] (similarly, QSQ [41]), incremental datalog evaluation (see [7]), and constraint databases [29]. Connections between tree-pattern queries and monadic datalog have been studied in [25]. In [38], datalog is extended with XPath predicates; the evaluation is not incremental. Connections between tree-pattern queries over trees and XPath expressions have been investigated; see e.g., [33].

Incremental view maintenance for a graph semistructured data is studied in [13]. Some recent works have addressed the issue of incremental maintenance of XPath views over trees [37, 39]. The maintenance of XQuery views is studied in [24] but without data streams and with data fully residing in memory.

Data stream processing has been intensively studied, in particular for the relational model, e.g. the Borealis [1], Aurora [2] and STREAM [34] systems. In the XML stream processing field, works on stream filtering like [19, 26] illustrate automata-based approaches for indexing path queries. Several XQuery processors for XML data streams have been proposed, e.g. [23, 31], as well as distributed systems that handle streams e.g. StreamGlobe [32, 40]. Some works, e.g. [22], are blending stream processing with optimization techniques for XML databases, however in a quite different setting.

Query evaluation for active documents is studied in [4]. The context is essentially different since the functions are non-stream and incremental maintenance not considered. Other works on active documents may be found at [14].

We have introduced a new concept, the axlog widget, which is a view over an active document. Its behavior is extremely simple: it has some input update streams and produces an output update stream (or more). We believe that this is an important notion both for specifying distributed computations and for supporting/optimizing them. In particular, we believe that axlog widgets will become important components of the AXML world. They already form the kernel of P2PMonitor, the distributed monitoring system for AXML.

The system is already running and has been demonstrated [12]. Many directions for improvements are opened. We mentioned the management of deletions and nonpositive queries. We also mentioned distributed optimization issues. When many "clients" subscribe to the same stream, broadcasting techniques may be considered. The notion of satisfiability seems the starting point of fascinating theoretical questions related to temporal logic. The notion of relevance seems to also open promising directions; we mentioned one, namely garbage collection in widgets.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2), 2003.

[3] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *PODS*, 2005.

[4] 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.

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

[6] Serge Abiteboul, Pierre Bourhis, and Bogdan Marinoiu. Satisfiability and Relevance for Queries over

Active Documents. http://www-rocq.inria.fr/∼marinoiu/satisfiability.pdf.

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

[8] Serge Abiteboul, Haim Kaplan, and Tova Milo. Compact labeling schemes for ancestor queries. In *SODA*, pages 547–556, 2001.

[9] Serge Abiteboul, Ioana Manolescu, and Emanuel Taropa. A framework for distributed XML data management. In *EDBT*, pages 1049–1058, 2006.

[10] Serge Abiteboul, Ioana Manolescu, and Spyros Zoupanos. Optimax: efficient support for data-intensive mash-ups. In *ICDE*, pages 1564–1567, 2008.

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

[12] Serge Abiteboul, Bogdan Marinoiu, and Pierre Bourhis. Distributed Monitoring of Peer to Peer Systems (demo). In *ICDE*, 2008.

[13] 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.

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

[15] Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *J. Log. Program.*, 4(3):259–262, 1987.

[16] Catriel Beeri and Raghu Ramakrishnan. On the Power of Magic. In *PODS*, pages 269–284, 1987.

[17] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *ACM SIGMOD Conference on Management of data*, 2006.

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

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

[20] Robert Ennals and David Gay. User-friendly functional programming for Web mashups. In *ICFP*, pages 223–234, 2007.

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

[22] Mary F. Fernández, Philippe Michiels, Jérôme Siméon, and Michael Stark. XQuery streaming à la carte. In *ICDE*, pages 256–265, 2007.

[23] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan, and Geetika Agrawal. The BEA/XQRL Streaming XQuery Processor. In *VLDB*, pages 997–1008, 2003.

[24] J. Nathan Foster, Ravi Konuru, Jérôme Siméon, and Lionel Villard. An algebraic approach to view maintenance for XQuery. In *PLAN-X*, 2008.

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

[26] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *ICDT*, 2003.

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

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

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

[30] Roman Kapuscinski, Rachel Q. Zhang, Paul Carbonneau, Robert Moore, and Bill Reeves. Inventory decisions in Dell's supply chain. *Interfaces*, 34(3):191–205, 2004.

[31] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, pages 1309–1312, 2004.

[32] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. Streamglobe: Processing and sharing Data Streams in Grid-Based P2P infrastructures. In *VLDB*, 2005.

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

[34] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[35] Wolfgang Nejdl, Stefano Ceri, and Gio Wiederhold. Evaluating recursive queries in distributed databases. *IEEE Trans. Knowl. Data Eng.*, 5(1):104–121, 1993.

[36] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD Conference*, pages 903–908, 2004.

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

[38] Royi Ronen and Oded Shmueli. Evaluation of datalog extended with an XPath predicate. In *WIDM*, pages 9–16, 2007.

[39] 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.

[40] Bernhard Stegmaier, Richard Kuntscke, and Alfons Kemper. Streamglobe: adaptive query processing and optimization in streaming P2P environments. In *ACM International Conference Proceeding Series; Vol. 72*, 2004.

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

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

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