Lazy Query Evaluation for Active XML*

Serge Abiteboul
INRIA Futurs & Xyleme Corp.
Serge.Abiteboul@inria.fr

Omar Benjelloun
INRIA Futurs
Omar.Benjelloun@inria.fr

Bogdan Cautis
INRIA Futurs
Bogdan.Cautis@inria.fr

Ioana Manolescu
INRIA Futurs
Ioana.Manolescu@inria.fr

Tova Milo
INRIA Futurs & Tel-Aviv U.
Tova.Milo@inria.fr

Nicoleta Preda
INRIA Futurs
Nicoleta.Preda@inria.fr

ABSTRACT

In this paper, we study query evaluation on Active XML documents (AXML for short), a new generation of XML documents that has recently gained popularity. AXML documents are XML documents whose content is given partly extensionally, by explicit data elements, and partly intensionally, by embedded calls to Web services, which can be invoked to generate data.

A major challenge in the efficient evaluation of queries over such documents is to detect which calls may bring data that is relevant for the query execution, and to avoid the materialization of irrelevant information. The problem is intricate, as service calls may be embedded anywhere in the document, and service invocations possibly return data containing calls to new services. Hence, the detection of relevant calls becomes a continuous process. Also, a good analysis must take the service signatures into consideration.

We formalize the problem, and provide algorithms to solve it. We also present an implementation that is compliant with XML and Web services standards, and is used as part of the ActiveXML system. Finally, we experimentally measure the performance gains obtained by a careful filtering of the service calls to be triggered.

1. INTRODUCTION

The increasing popularity of XML and Web services [28] has recently promoted XML documents with embedded calls to Web services as a useful paradigm for distributed data management on the Web [1, 25, 17, 18]. In the line of [1], we refer to them as *Active XML* (AXML) documents. More precisely, AXML documents are XML documents where some of the data is given explicitly, while other parts are given only intensionally, using special XML elements that are interpreted as calls to Web services. When the calls are invoked, their results are inserted in the document. To answer a query on an AXML document, one would like to be "lazy", in the sense of avoiding the invocation of service calls that bring data not relevant for the query. The main contribution of this paper is an algorithm for the lazy evaluation of queries on AXML documents.

The original AXML system [1] supports many features to control the activation of service calls embedded in documents. For example, a particular service call may be invoked at regular time intervals or only upon explicit user intervention. We are concerned here with a special kind of call activation: *lazy service calls*. A service call specified as *lazy* is invoked only when its result may participate in the answer to a pending query. To illustrate this notion, consider a Web site about your city's night-life [1], which is described by an AXML document, containing information about, say, movies and restaurants. Now, suppose someone asks the query `/goingOut/movies//show[title="The Hours"]/schedule`. Then, there is no point in invoking any calls found below the path `/goingOut/restaurants`, since the data they return would be set in a place where they cannot contribute to the result. One would also like to avoid invoking calls found under `/goingOut/movies` whose signature indicates that the returned data is irrelevant to the query, e.g., service calls returning movie reviews.

The previous observation rules out a naive approach that consists in invoking *all* the calls in the document recursively, until a fixpoint is reached, and finally running the query over the resulting document. The query processing needs to be more sophisticate and in particular to call services selectively. A less naive approach is to build a query processor that traverses the document top-down, and invokes the calls encountered while evaluating the query. This materializes only part of the document – the subtree traversed by the top-down evaluation. However, the mixing of query processing and service invocation would result in bad performances, since the query processor would either have to be blocked waiting for call responses, or would have to be restarted several times to account for the document growth. Moreover, forcing a particular query processing strategy eliminates the opportunities for query optimization. Instead, the technique we propose consists in identifying a tight superset of the service calls that should actually be invoked to answer a query. The idea is to first invoke these calls and then move to a classical query processing as a second stage.

Observe that while the problem we address is close to the mediation paradigm, where data sources are called to answer queries on a mediated schema, things are substantially different in AXML. Service calls may appear (i) *anywhere*

*This work was partially supported by EU IST project DBGlobe (IST 2001-32645).

SIGMOD 2004 June 13-18, 2004, Paris, France.

[1] In the style of http://www.timeout.com

in the *data*, and (*ii*) *dynamically* in results of previously invoked calls. Moreover, the relevance of one call may *depend* on the result of another one, and the service calls *return types* should be taken into consideration. A first contribution of the paper is a better understanding of the possible relationships among the service calls embedded in a document, and their influence on the relevance of calls to queries. Based on that, our central contribution is an efficient algorithm for lazy query evaluation, whose key facets are:

- **Computing the set of relevant service calls.** The algorithm generates a set of queries that retrieve all service calls relevant because of there position.
- **Service calls sequencing.** Relationships among the calls are analyzed, to derive a sequence of call invocations appropriate to answer a query.
- **Pruning via typing.** Return types of services are used to rule out more service calls.
- **Service calls guide.** A specialized access structure is used to speed up the detection of relevant calls.
- **Pushing queries.** Precise knowledge of the interaction between the query and each service call enables pushing queries to capable Web services, like mediators do with data sources.

The algorithm we propose is *dynamic*, in the sense that it adapts to the state of the document, as this state is modified by service invocations, and decides at each point which services should be invoked next. It has been implemented in the ActiveXML system [1], which supports the AXML model. Our experimental results, presented in Section 8, show that, compared to the naive approach, the pruning of irrelevant service calls may reduce the overall query evaluation time by orders of magnitude. They also demonstrate the gain obtained from pushing queries to service providers.

A recent work [21] addressed the issue of call invocations for exchanging AXML documents. There, the goal was to identify the calls that need to be invoked for making a document match an XML schema, and the solution relied on schema analysis via automata-based algorithms. By contrast, our goal here is to find the sequence of call invocations needed to evaluate a query over AXML data, and the solution is based on query analysis and rewriting. Indeed the two techniques are complementary: The techniques of [21] can be used to ensure that the parameters of calls invoked in our query processing match the type required by the service signature, and can also be applied to query results if they need to be exchanged. Conversely, our technique can be used to evaluate queries on exchanged AXML data.

The paper is organized as follows: Section 2 describes the AXML data model and query language, and formalizes the general problem. Section 3 explains how to find, for a given state of a document, the service calls currently relevant for a query. Then, using a delicate analysis of the relationship among the calls, we provide in Section 4 an efficient algorithm to derive the sequence of call invocations needed to answer a query. A refinement, based on the analysis of service signatures is described in Section 5. Two techniques to further speed up the computation, via query relaxation and a special access structure, are described in Section 6. The pushing of queries is considered in Section 7. The implementation and experimental results are described in Section 8. The last section studies related works and concludes.

2. PRELIMINARIES

To define the problem formally, we start by briefly presenting a simplified view of the AXML data model, borrowed from [21]. We describe the query language we use, which captures the core tree-pattern matching fragment of XQuery. Then, we introduce the new notion of *relevance* of service calls for queries, which will be central to the lazy query evaluation considered in further sections.

Documents and services. AXML documents are modeled as ordered labeled trees with *data* and *function* nodes. The data nodes represent the regular XML parts of a document, and are labeled with element names or data values (for leaves). The function nodes represent calls to Web services, and are labeled by function names [2]. The children subtrees of a function node are the *parameters* of the call. When the function is called, these subtrees are passed to it. The return value then replaces the function node in the document. A sample AXML document is shown in Figure 1. Function nodes have bold labels, and are numbered so that they can be referred to in later discussions. Data values are quoted.

This document contains a list of hotels, some of which are given explicitly and some only intensionally, through an embedded call to the `getHotels` function. The document details (extensionally or intensionally) for each hotel its name, address, rating, and some nearby restaurants or museums. When the first `getNearbyRestos` call is invoked with the address of the hotel as a parameter, it returns a list of `restaurant` elements, which replaces the function node, as shown in Figure 3 (ignore for now the gray marking). Note that the parameter subtrees and the return values may themselves be AXML documents.

Signatures of Web services, namely the expected type of their parameters and results, are typically given in their WSDL description [28]. For services with AXML input or output, these types also entail information about the intensional parts of the data, and detail which service calls may appear where [21]. For simplicity, we use here the same DTD-like syntax as in [21]. Signatures are described by a schema τ that associates (1) a pair of regular expressions with each function name f, which respectively describe the input and output type of the function, and (2) a regular expression with each element name, which describes the structure of the element. The keyword `data` is used for data values. A sample schema τ can be seen in Figure 2.

Consider a particular function node. Its input must be properly typed, i.e. the labels of its children must match the corresponding `in` regular expression. Its result is guaranteed to match the `out` regular expression. Like in DTDs, for a data node occurring in an input/output subtree, the regular expression associated with its label must be matched by the labels of its children. It is easy to see that the output of `getNearbyRestos` in Figure 3 matches the corresponding output type in τ.

Queries. Queries are modeled by *tree patterns*. A tree pattern query is a labeled tree whose nodes are labeled by variable names, constants (element names and data values), or the special symbol $*$. The nodes labeled by variable names

[2]In practice, the function name corresponds to the parameters that identify the Web service: its URL, namespace, etc.

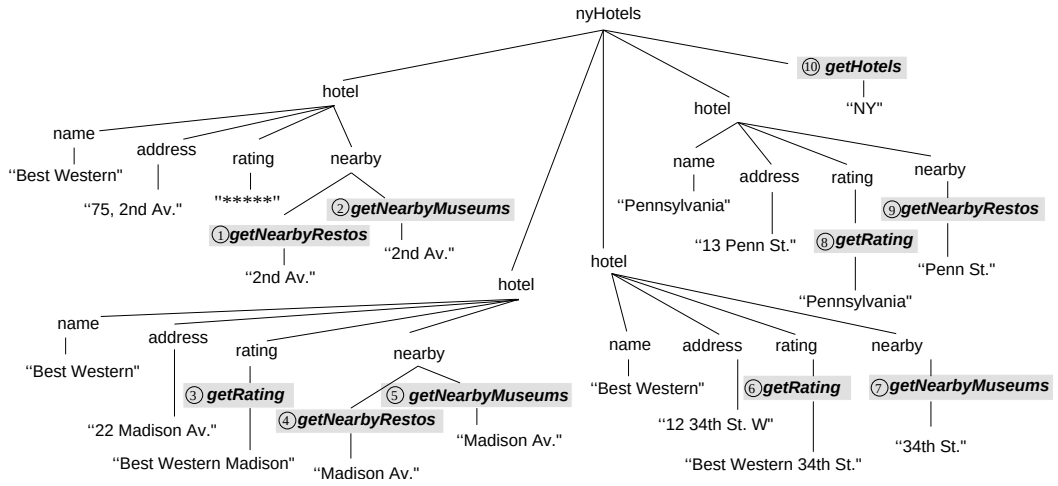

Figure 1: A sample Active XML document

functions:

getHotels	=	[**in:** data,**out:** hotel*]
getRating	=	[**in:** data,**out:** data]
getNearbyRestos	=	[**in:** data,**out:** restaurant*]
getNearbyMuseums	=	[**in:** data,**out:** museum*]

data:

hotel	=	name.address.rating.nearby
nearby	=	restaurant*.getNearbyRestos* .museum*.getNearbyMuseums*
restaurant	=	name.address.rating
museum	=	name.address
name	=	data
address	=	data
rating	=	(data | getRating)

Figure 2: A schema τ for function signatures.

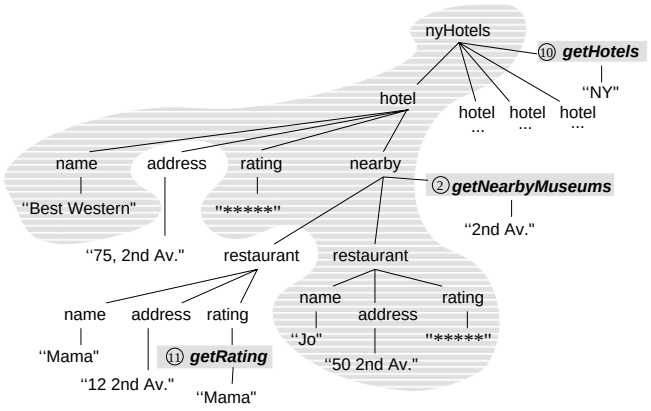

Figure 3: After invoking getNearbyRestos

are called *variable nodes* and those labeled by element names and data values are called *constant nodes*. The tree also has a distinguished set of edges called *descendant edges* and a distinguished set of nodes called the *result nodes*.

Figure 4 shows a tree pattern query. Descendant edges are represented by double lines and output nodes are pointed by little arrows.

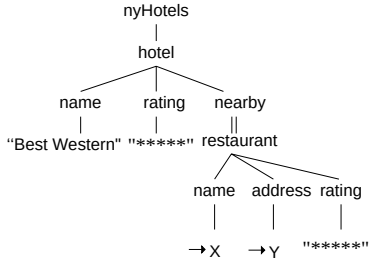

Figure 4: A tree pattern query

To define the semantics of queries, we use the notion of query embedding.

Definition 1. *Given a tree pattern query q and an AXML document d, an* **embedding** *of q into d is a tree homomorphism ρ from the nodes of q to the* data nodes *of d, mapping the root of q to that of d, preserving the parent-child and ancestor-descendant relationships (for regular and descendant edges, resp.), mapping each constant node of q to a data node of d with the same label, and such that all the variable nodes of q with the same variable name are mapped to data nodes having identical labels.*

The restriction of ρ to the result nodes of the pattern q is called the result *of the embedding. The set of results of all possible embeddings is called the* **snapshot result** *of the query, and denoted $q(d)$.*

Note that the data nodes of d that *contribute* to a given embedding are the images of the pattern nodes, plus the nodes in d residing on the paths between those images (i.e. d nodes that correspond to descendants edges in the pattern). Consequently, we say that a data node in d **contributes** to $q(d)$ if it contributes to some embedding of q in d.

For instance, the query of Figure 4 has no embedding in the document of Figure 1 and the snapshot result on this document is thus empty. On the other hand, once the `getNearbyRestos` function is invoked, the query has one embedding in the document of Figure 3. The marked grey area contains the nodes that contribute to the embedding, and the snapshot result on this document consists of the name and address of the marked restaurant.

Observe that, while AXML documents are ordered, our tree pattern queries are not sensitive to document order. This is because the former represent XML data (which is ordered), while the latter are intended to capture the core of XPath/XQuery, focusing on vertical axes like child and descendant, where the order of elements is irrelevant.

It is also important to note that queries only match the *data nodes* of the document. This is because function nodes are only means to get the data they represent. Consequently, we distinguish between two possible semantics for a query: The ***snapshot result***, defined above, is the result of the query when evaluated on the document *in its current state*, without invoking the function calls it contains. By contrast, the ***full result*** of a query represents (possibly intensionally) its snapshot result when evaluated on the document in its "full" state. Intuitively, by full state we mean the state in which all possible calls in the document have been evaluated, i.e., the document is fully materialized. The notions of full state and result are formalized in [2]. *In a nutshell, this paper focuses on efficiently computing the full result of a query, while intelligently invoking as few function calls as possible.*

Relevant calls.

When functions calls are invoked, their answers, inserted in the document, may contribute new embeddings to the query. As the document evolves, the snapshot result of the query may grow. We would like to (*i*) invoke only function calls that may indeed contribute to this growth, and (*ii*) detect when no more function invocations can add significant data, and the full query result can be computed by simply evaluating the query on the document. These notions are formalized next.

DEFINITION 2. *For an AXML document d_1, we say that $d_1 \xrightarrow{v} d_2$ if d_2 is obtained from d_1 by selecting a function node v in d_1 with some label f and replacing it by an arbitrary output instance of f* [3]*. If $d_1 \xrightarrow{v_1} d_2 \xrightarrow{v_2} d_3 \ldots \xrightarrow{v_{n-1}} d_n$ we say that d_1* **rewrites** *into d_n, denoted $d_1 \xrightarrow{*} d_n$.*

We say that a node $v \in d_n$ was **produced by** *a function node v_i if $v \notin d_i$ and $v \in d_{i+1}$. v is* **transitively produced by** *v_i if it is either produced by v_i or is produced by some function node v_j transitively produced by v_i.*

Note that in the rewriting process, the replacement of a function node v by an output instance is independent of any function semantics. In particular, we may replace calls to the same function by different output instances. This captures the behavior of real life Web services, like a temperature or stock exchange service, for which two calls may yield different results.

[3]By *replacing the node by an output instance* we mean that the node v and the subtree rooted at it are deleted from d_1, and the forest trees $d'_1, \ldots, d'_j$ of some output instance of f are plugged in place of v.

DEFINITION 3. *Given a document d, we say that a function node $v \in d$ is* **relevant** *for a query q if there exists some rewriting $d \xrightarrow{v} d_1 \xrightarrow{*} d_n$ where some of the data nodes transitively produced by v contribute to $q(d_n)$.*

If the document contains no relevant calls, it is said to be **complete** *for the query.*

Observe that the notion of function relevance has an *optimistic* nature: a function is considered relevant if it is possible that it returns a "good" result which, possibly together with additional "good" data produced by the invocation of other functions (in its own output or in other places in the document) will contribute to the query evaluation.

To illustrate, consider the document of Figure 1 and the query of Figure 4. The relevant functions here are 1, 3, 4 and 10. Function 1 is relevant, as it may produce restaurant elements with a high rating, or a `getRating` function call that may produce such a rating, and hence make the hotel qualify for the query. Function 4 is relevant, as it may similarly produce highly rated restaurants, which, together with a potential high rating for the hotel, returned by function 3, may also make this hotel qualify. Function 3 is thus relevant for symmetric reasons. Function 10 may return qualifying hotels. Other function calls are irrelevant, either because their output is of a type that cannot contribute to the query (e.g. 7) or because, regardless of their answer, the corresponding hotel cannot satisfy the query criteria (e.g. 6 and 8).

Relevant rewritings.

Our goal is to invoke relevant calls in the document, until it is complete for the query, and then evaluate the query on it. An immediate issue that arises is *termination*: since function invocations may return new data and new function calls, a rewriting may never terminate. This behavior is inherent in the AXML model, and is carefully studied in [2], which provides sufficient conditions for termination. We will not consider this issue here, and assume that either these conditions hold or that the computation halts if a full state is not reached after some time limit. We focus here on the selective invocation of functions and completeness detection.

Note that once a function is invoked and returns a specific answer (one among all the possible instances of the function's output type), other functions that were considered relevant before may cease to be so. For instance, if function 3 is invoked and returns a low rating, function 4 is no longer relevant. This motivates the following definition.

DEFINITION 4. *For a document d_1 and a query q, we say that a rewriting $d_1 \xrightarrow{v_1} d_2 \xrightarrow{v_2} d_3 \ldots \xrightarrow{v_{n-1}} d_n$ is* **relevant** *if $\forall i \in 1 \ldots, n-1$, the function node v_i is relevant for d_i and q. The rewriting is* **complete** *if d_n is complete for q.*

In a relevant rewriting, only functions that may contribute to the query result are invoked. This is in contrast to the naive approach where all functions are invoked. Note however that there is a tradeoff between *accuracy* and *efficiency*: if it is expensive to exactly detect which calls are relevant and which are not, one may prefer a more lenient rewriting, where all relevant calls are indeed invoked but perhaps also some additional, non relevant ones. Note that such a rewriting is "safe", in the sense that it does not change the query result. The challenge is thus to find the right balance between the efforts spent on ruling out irrelevant calls and the actual time saved by avoiding their invocation.

Pushing queries. Even when a call is relevant, its *entire* result may not be needed to evaluate the query. For instance, `getNearbyRestos` may return many restaurants. As we are only interested in five-star ones, and more precisely, only in their names and addresses, we may want to *push* to the function call a precise *subquery*, specifying that it has to apply the five-star rating selection, and only return the relevant names and addresses.

The results. We can now state more formally the contributions of this paper.

1. We present an algorithm that, given a query q and a document d, finds all the function calls in d that are relevant for q.
2. The above algorithm induces a simple mechanism to find a relevant rewriting, that iteratively invokes relevant calls, and applies the previous algorithm at each step on the changed document. We improve on this, by analyzing dependencies among relevant functions, and minimizing the needed efforts to find, at each step of the rewriting, the next function to be invoked.
3. We analyze the complexity of our algorithms, and propose two complementary methods to improve performance. The first uses a lenient variant of the algorithms, with a less refined but more efficient call relevance detection component. The second is based on an access structure to speed up processing.
4. We provide an algorithm that determines the subqueries to be pushed to the function calls that we decide to trigger, and a simple execution model for controlling this distributed computation.
5. Finally, we describe our implementation and provide an experimental assessment of the performances of the above techniques.

Some useful machinery. Before presenting our results, we need to introduce two extensions to the query language, that will be useful in the sequel. We will use two new types of nodes in our tree pattern queries:

The first are *`OR` nodes*, that represent a *choice* between their children subtrees. A query q with `OR`-nodes can be viewed as the union of all queries q_i without `OR`-nodes that can be obtained from q by some choice of a single child subtree for each `OR`-node in q.

The second are *function nodes*. The queries that we saw so far were concerned only with data nodes. For queries with function nodes, the snapshot result is defined as for regular queries, except that the pattern embeddings map the function nodes in the query to function calls in the document.

We refer to tree patterns having `OR` and $function$ nodes as ***extended*** queries. Examples are given in Figure 6. `OR`-nodes are labeled by $\oplus$ and function nodes have a gray background. In the following sections, we will use such extended queries to retrieve relevant function calls.

For compactness, rather than drawing tree patterns, we will sometimes use an XPath-like syntax for queries. In this syntactic representation, to distinguish data nodes from function nodes, we will add parentheses to the latter, as in $nyHotels/hotel/rating/getRating()$.

3. FINDING RELEVANT CALLS

Given a document d and a query q, our goal is to find a complete relevant rewriting, that is, a sequence of function invocations that makes d complete for q, and such that, at each step, the invoked function call is relevant w.r.t. the current state of the document. For simplicity, we will first ignore the functions signatures and assume that functions can return arbitrary answers, i.e. that their output type is `any`. We will see in Section 5 how to use function signatures.

We start by explaining how to find, for a given state of the document, the function nodes currently relevant to q. We present two algorithms, starting with the simpler one, and compare their respective properties. Based on that, we then provide algorithms to find a complete relevant rewriting.

3.1 Linear path queries (LPQ)

The portion of a document that can be modified by the activation of a function is clearly determined, since the result is placed at the exact position of the function node. Therefore, a function node may be relevant for a query q only if it is on a path traversed by q. Based on this observation, we build a family of *extended queries* to retrieve relevant calls as follows: First, construct the *linear path queries* (LPQs) "embedded in q", i.e. the linear sub-queries of q. Then, replace their last node with a star-labeled function node, and mark this node as the output node of the query.

For instance, using the XPath-like syntax introduced in Section 2, the LPQs derived for the query of Figure 4 are:

```
/*()
/nyHotels/*()
/nyHotels/hotel/*()
/nyHotels/hotel/name/*()
/nyHotels/hotel/rating/*()
/nyHotels/hotel/nearby/*()
/nyHotels/hotel/nearby//*()
/nyHotels/hotel/nearby//restaurant/*()
/nyHotels/hotel/nearby//restaurant/name/*()
/nyHotels/hotel/nearby//restaurant/address/*()
/nyHotels/hotel/nearby//restaurant/rating/*()
```

The star-labeled function nodes (denoted $*()$) at the end of each query retrieve the function nodes residing at the end of a matching path in the document. It is fairly straightforward to see that, for any document d, *any* call in d that is relevant for q is necessarily returned by one of these linear path queries, when evaluated on d. However, what the LPQs actually compute is a *superset* of the relevant function calls. To see that, notice that on the sample document of Figure 1, the LPQs above select, among others, the calls to `getRating` and `getNearbyRestos` for the `"Pennsylvania"` hotel. These calls are in fact irrelevant, since the query is only interested in hotels named `"Best Western"`.

3.2 Node-focused queries (NFQ)

We need to use more sophisticated queries to get better accuracy. For that, we introduce *node-focused queries (NFQs)*: Instead of constructing one linear path query per node in the query, we derive a (potentially more precise) NFQ, that includes the *filtering conditions* from the original query. Notice that the latter can either be satisfied by data present in the document, or by data produced by the invocation of function calls. Thus, the NFQ for a given query node v will ask for all function calls found on the same path as v, such that all the filtering conditions could be satisfied either by some data, or by some (future) function call result.

	NFQ(query q)
1	let $q_\oplus$ by a copy of q
2	**for each** node u in q
3	let u' be its counterpart in $q_\oplus$
4	replace u' in $q_\oplus$ by an OR between u' and f_u,
5	where f_u is a function node labeled $*$
6	**end for**
7	let $Q = \emptyset$
8	**for each** node v in $q_\oplus$
9	compute q_v as $q_\oplus - \{v$ and its subtree$\}$,
10	with f_v as output node
11	simplify q_v by removing redundant ORs,
12	add q_v to Q
13	**end for**
14	return Q

Figure 5: Building NFQs for a query q.

The algorithm building all NFQs for a given query q is depicted in Figure 5. It starts by constructing a "modified query" $q_\oplus$, in which all nodes u are replaced by a choice between u and a star-labeled function node. Then, for each node v of q, the NFQ of v, denoted q_v, is obtained from $q_\oplus$ by erasing v and its subtree, and by marking the function call sibling of v as a return node.

Figure 6 shows three of the NFQs for the query of Figure 4: (a), (b), and (c) retrieve the function calls that might return relevant hotels, restaurants, and hotel ratings respectively.

Notice that, in these NFQs, `OR` nodes appearing in $q_\oplus$ on the path from the query root to the output node were removed, together with their `*()` branch. This is because these branches cannot contribute bindings to the query result (and being under an `OR`, are not existentially required). Therefore, we can omit these nodes (in step 11 of the algorithm) without changing the semantics of NFQs.

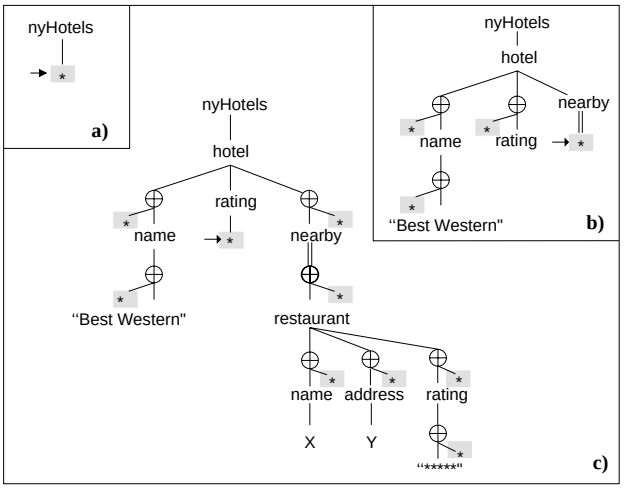

Figure 6: Three node-focused queries

The next property follows from the above discussion.

PROPOSITION 1. *Assuming that functions can return data of arbitrary type, the function nodes in a document d that are relevant for a query q are precisely those retrieved by the NFQs of q, when evaluated on d.*

4. SEQUENCING RELEVANT CALLS

We next explain how to derive a relevant rewriting, that is, a sequence of invocations in which *only relevant calls* are invoked. This is often an essential criterion, e.g., if we have to pay for calls. We will consider later some relaxations that simplify the analysis, at the cost of firing more calls than necessary.

4.1 The NFQA algorithm

NFQs induce a simple iterative algorithm for deriving a relevant rewriting: Given a document d_0, run the NFQs on d_0, pick *one* relevant function call f_0 among the returned calls, and invoke it, i.e. $d_0 \xrightarrow{f_0} d_1$. The next iteration will then run the NFQs *on the changed document* d_1, pick a relevant call f_1 among the results and invoke it, and so on recursively. We stop when no more calls are relevant, i.e. if for some step n, all NFQs return an empty answer on document d_n. We call this simple algorithm **NFQA** (for NFQ Algorithm). The next property follows:

PROPOSITION 2. *The NFQA algorithm computes a (possibly infinite) relevant rewriting. If it terminates, the obtained document is complete for the query q.*

Note that the potential non-termination is not due to the NFQA algorithm, but to the semantics of AXML documents/queries, that can indeed be infinite (see Section 2).

Clearly, through the execution of the NFQA algorithm, the NFQs stay the same, as they only depend on the query q. However, the *result* of the NFQs may change as the document evolves, thus they have to be re-evaluated at each iteration. The NFQs result may change in two ways:

1. The invocation of a call may bring new relevant calls, hence enrich the result of the NFQs.
2. Also, the invocation of a call may affect the document in such a way that formerly relevant calls are not relevant anymore, and this is what makes reevaluating the NFQs essential. For instance, if the `getRating` call of the `"Best Western"` hotel in Figure 1 is invoked and returns a low rating, the `getNearbyRestos` call, which was relevant before, becomes irrelevant.

To avoid the costly reevaluation of NFQs after each call, we analyze their mutual relationships. We present two optimizations, corresponding to items 1 and 2 above.

First, we characterize (in Section 4.2) the cases where the invocation of relevant calls, found by one NFQ, may return new relevant calls, enriching the result of another NFQ. This will enable us to apply a layering strategy, that splits the set of NFQs into smaller groups, on which NFQA is applied separately. Running NFQA on smaller groups *may yield much less* NFQ evaluations than doing so on the initial set.

Second, we detect some useful cases where a given call, once known to be relevant, is guaranteed to stay so (in Section 4.4). Assuming this property holds for two calls f_1 and f_2, we can invoke f_1 and f_2 *in parallel*, since we are sure that triggering one will not affect the relevance of the other.

Other optimizations are possible, since relevant rewritings are found by issuing NFQ *queries*. Their evaluation is delegated to a query processor, which can apply standard query optimization techniques [10], or eliminate redundant queries using containment checking as in [20]. The benefits

of such techniques naturally apply in our setting. In particular, techniques for multi-query optimization [7] are essential to avoid performance penalties. In this section, we focus on techniques that are specific to our context.

4.2 Influence of NFQs

For a tree pattern query q and a node $v \in q$, we denote by q_v the NFQ of node v. For two nodes $v, v' \in q$, we say that q_v *may influence* $q_{v'}$ if the invocation of some call retrieved by q_v may bring new calls, detected by q'_v. More formally,

DEFINITION 5. *For two nodes $v, v' \in q$, the NFQ q_v* **may influence** *$q_{v'}$ if there exist a document d, a function node $u \in q_v(d)$, and a rewriting $d \xrightarrow{u} d_1 \xrightarrow{*} d_n$, such that $q_{v'}(d_n)$ contains some function node transitively produced by u.*

For instance, the NFQ of Figure 6(a) may influence the ones of Figures 6(b) and (c), as the `getHotels` it retrieves may return, a `hotel` element with `getRating` or `getNearbyMuseum` calls in the right positions in its subtree.

We now explain how to detect whether an NFQ may influence another one. Recall that the result of a function call is placed in the document at the exact position of the call. Consequently, for an NFQ q_v to be able to influence another NFQ $q_{v'}$, the function calls selected by q_v should be *higher* in the tree than those selected by $q_{v'}$, so that the output of the former can actually be traversed by $q_{v'}$.

To state this formally, we use the auxiliary notion of *linear part of an NFQ* q_v, denoted q_v^{lin}, which is the linear path expression present in q, going from the root to v (not included). For example, q_v^{lin} for the NFQ of Figure 6(c) is `/nyHotels/hotel/rating`. Our algorithm for testing potential influence among NFQs is based on the following observation:

PROPOSITION 3. *For a query q and two nodes $v, v' \in q$, the NFQ q_v* may influence *the NFQ $q_{v'}$ iff some word in the regular language of q_v^{lin} is a prefix of some word in $q_{v'}^{lin}$.*

This proposition gives an immediate PTIME algorithm to check the influence relationship between NFQs: (1) build two regular automata, one accepting the words in q_v^{lin} and one accepting *the prefixes* of words in $q_{v'}^{lin}$, (2) build their cartesian product automaton, which accepts the intersection of the two languages [16], and (3) test if the latter is empty.

4.3 NFQ layers

We use the *may influence* relation among NFQs to split them into smaller groups, called *layers*.

Let $\leq$ denote the transitive closure of the *may influence* relation, and let $\equiv$ be the equivalence relation defined by $q_v \equiv q_{v'}$ if $q_v \leq q_{v'}$ and $q_{v'} \leq q_v$. Let NFQ *layers* be the equivalence classes of $\equiv$. $\leq$ induces a *partial order* between layers. To find a relevant rewriting, the interactions between NFQs can be analyzed based on their respective layers.

- By construction, NFQs inside one layer may influence each other, therefore, we need to run NFQA, as described in Section 4.1, inside each layer.
- For two *comparable* layers L_1,L_2, where $L_1 \leq L_2$ [4], we know that the processing of L_1 may augment the result of the queries in L_2, but the reverse is not true. Thus, we process L_1 first, and then we can process L_2.

[4]By abuse of notation, $\leq$ is also used for layers.

This entails the following evaluation strategy: The partial order among layers is completed into some compatible total order. Layers are processed in increasing order. Within each layer, we apply NFQA. Note that when the processing of a given layer is over, we can simplify the remaining NFQs by removing the `OR/*()` branches corresponding to the layer we just finished. This is because the corresponding functions have already been invoked and cannot appear in the document anymore. This simplification makes the NFQs much less complex, hence faster to evaluate, without changing their results.

In our running example, the NFQ of Figure 6(a) may influence those of Figures 6(b) and 6(c), which are incomparable. Consequently, each layer consists of a single NFQ, with the one of Figure 6(a) being the first.

In general, layers may contain several NFQs. For instance, two NFQs $q_v, q_{v'}$ with linear paths q_v^{lin} = `//a` and $q_{v'}^{lin}$ = `//b` would belong to the same layer, as paths that end with a `b` may have a prefix that end with an `a` and vice versa.

4.4 Parallelizing calls

The layering process described above decomposed the computation into smaller independent chunks. Let us now see how to further optimize this processing, by parallelizing the computation, *inside* one layer. More precisely, we want to know, given an NFQ q_v, under which condition relevant function calls returned by q_v are guaranteed to stay relevant, independently of calling the other relevant function calls returned by q_v. In this case we can invoke all the returned calls in parallel and spare the re-evaluations of q_v that the naive NFQA algorithm needed after triggering each call.

This is the case, for example, for NFQs q_v satisfying condition (∗) below, which we call the *independence* condition. As before, q_v^{lin} here denotes the linear path expression going from the root of q to v, not included.

(*) For each NFQ $q_{v'}$ in the same layer as q_v, $q_v^{lin} \cap q_{v'}^{lin} = \emptyset$.

We omit the proof for space constraints and only note that this condition can be easily checked by constructing the regular automata of the two path expressions, and checking that their cartesian product is empty [16].

Going back to our running example, as each of the layers here contains a single NFQ, the NFQ is trivially independent and all the calls it retrieves can be invoked in parallel. The NFQA algorithm will then reevaluate the NFQ, to check if new calls were added by the previous invocations, and if so, will invoke the new calls in parallel, an so on. As another example, we saw a layer with two NFQs at the end of the previous subsection. As the intersection of their linear paths q_v^{lin} = `//a` and $q_{v'}^{lin}$ = `//b` is empty, both NFQs are independent. For this layer the NFQA algorithm will, repeatedly, pick one of them, evaluate it and invoke all the retrieved functions, until no more functions are found.

We focused here on the maximum parallelism that can be achieved *without performing unnecessary calls*. Note that one may be able to reduce the time it takes to produce the answer by calling functions in parallel *just in case*, and thereby introduce more parallelism. This is an interesting research direction that requires the use of a cost model, and will not be considered here for space reasons.

5. USING TYPES

So far, our analysis ignored function signatures. Let us now see how to use them to rule out more irrelevant calls.

The key idea guiding the construction of NFQs in the previous section was that, for each subtree of the query q, documents may either contain the subtree explicitly, or contain a function call that may return it. In practice, the possible shape of the data returned by a function is determined by its output type, and the output types of the functions that may appear in its output, recursively. To verify that the returned data may indeed be of the shape required by the query, the functions signature needs to be analyzed.

DEFINITION 6. *Given a schema τ, a function f defined in τ* **satisfies** *a query q if $q(d) \neq \emptyset$ for some* derived instance *d of f's output type, where the* **derived instances** *of a type are all the documents that instances of the type can be rewritten into.*

We will explain below how, given a function f and a query q, one can test if f *satisfies* q. For now, let us simply assume that such a satisfiability detection algorithm is given, and see how it can be used to refine the NFQs of Section 3.

Refined NFQs. We use the following notation: For a query q and a node $v \in q$, sub_v^q denotes the query subtree rooted at v. We will also use, as previously, q_v to denote the NFQ query of v, and $q_v(d)$ to denote the result of q_v when evaluated on document d.

Recall that, for each node v, q_v looks for the functions that may contribute to the subtree pattern sub_v^q rooted at v. A natural filtering that one can apply to the functions retrieved by $q_v(d)$, is to discard those that do not satisfy sub_v^q: since these functions cannot contribute to the part of the query for which they have been selected, there is no point in invoking them. We can, for instance, discard all the `getNearbyMuseums` retrieved by the NFQ of Figure 6(b), since they return museum elements, and hence cannot satisfy the subquery `//restaurant[name=X,address=Y,rating=''*****'']`.

Note however that, while the above analysis can discard some irrelevant calls, it may still let irrelevant ones go through. Consider for instance the `getRating` call numbered 6 in Figure 1, which is retrieved by the NFQ in Figure 6(c). The function has a "correct" output type, but it is irrelevant, since the hotel's *nearby* element contains a function with inappropriate output type. To take this kind of typing information into consideration, the NFQs need to be *refined*, and must explicitly detail, for each query node v, which functions are "eligible", in the sense that they may actually produce data matched by the subquery sub_v^q rooted at v.

More precisely, given the list of all the function names retrieved by NFQs, we can test for each function f and each node v if f satisfies sub_v^q. Then, *refined NFQs* can be constructed as in the algorithm of Figure 5, except that rather than `OR`-ing each node $v \in q$ with a starred function node $*()$ matching any function, only the concrete names of the functions satisfying sub_v^q are listed. *These refined NFQs retrieve precisely the set of functions relevant for q and d.*

For instance, the refined version of the NFQ of Figure 6(c) is depicted in Figure 7. Note that some of the function nodes have been removed (as there were no functions producing data of that type) and the $*()$ labels have been replaced by concrete function names.

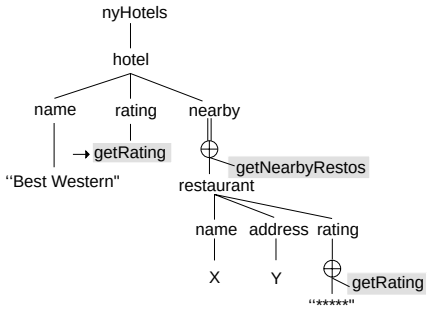

Figure 7: Refined NFQ

A similar refinement can also be applied to the layering and parallelism processes of Section 4. Details are omitted.

Note that during the rewriting process, when functions are invoked, their result bring new function calls into the document. The refined NFQs are enriched accordingly: the names of the new functions are added as possible alternatives to the query subtrees that they satisfy.

Testing satisfiability. To conclude the discussion, we need to explain how, given a function f and a (sub)query q, one can test if f satisfies q. Due to space limitations we only sketch the main ideas of our algorithm.

To check satisfiability, one needs to test if $q(d) \neq \emptyset$ for some derived instance of f's output type. Observe that a similar problem has been studied, e.g. in [22], for regular (non Active) XML documents: given a schema and a tree pattern query q, Milo and Suciu provide an algorithm for testing if $q(d) \neq \emptyset$ for an instance d of the schema. The difference here is that we are interested in testing satisfiability for the *derived* instances of the function output type, and not for its direct instances. By extending the algorithm of [22] to deal with derived instances, we obtain an algorithm for testing function satisfiability, that runs in time exponential in the size of the schema and the query. One can further show that the problem is NP-hard even for very simple queries. So it is unlikely that an algorithm with a lower time complexity exists. While this may seem negative, observe that the complexity is not w.r.t the data size, but w.r.t the size of the schema and query, which are likely to be much smaller. Nevertheless, to further reduce the running time, an alternative is to use some lenient sufficient conditions, that may qualify more functions than actually needed, but can be tested more efficiently. We will see such a lenient algorithm, used in our implementation, in Section 6.1.

6. FASTER RELEVANCE DETECTION

In this section we propose two complementary techniques to speed up the detection of relevant calls.

6.1 Lenient rewriting

The algorithm presented above guarantees that only relevant functions are invoked. To reduce the analysis time, one can trade *accuracy* for *efficiency*, running somewhat more lenient (but faster) analysis, that invokes all relevant calls but possibly some more. In particular one can (a) relax the queries that find relevant function calls, and (b) relax the analysis of functions signatures.

Relaxed NFQ. A substantial efficiency gain can be obtained by "approximating" each NFQ by a simpler query that can be evaluated faster. Two languages are natural candidates to approximate NFQs. The first one is *XPath*. XPath queries can express all variable-free tree patterns. Approximating an NFQ by an XPath expression thus ignores the value-based joins. In return, the XPath approximation has lower time complexity and can be processed faster [12]. The second is *linear path queries*: the LPQs introduced in Section 3 can be seen as a relaxed version of NFQs, accounting only for their linear part. Note that the NFQ layering and parallelism techniques presented in Section 4 can be used without modification for these relaxed queries.

Relaxed schema. A crucial part of the analysis of function signatures is testing which sub-queries they can satisfy. To speed up this test, we use in our implementation a lenient description of the output types of functions, which ignores the cardinality of elements and their order. The derived output type of a function is then represented by a simple *graph schema*, in the spirit of [8], and checking satisfiability amounts to checking if the query can be embedded in this graph. This can be tested in time polynomial in the size of the schema. We omit the details.

A lenient relevance analysis can also be used as a preprocessing step, doing a first filtering of the irrelevant service calls before the "exact" analysis is applied. The benefits of such preprocessing are illustrated next.

6.2 Function call guides

To further speed-up the processing, we use an access structure that provides information about the function calls that are present in the document. For efficiency, this information should be compact, and available in a way that is easily exploitable by the query processor.

In the spirit of dataguides [11], we use a tree structure that summarizes the paths that occur in a given document, containing a single occurrence of each path. The originality here is that we limit ourselves to paths leading to function calls. We call this structure a *function call guide* (F-guide for short). The F-guide also holds the path *extents*: for each path we keep pointers to the corresponding function call nodes in the document. Figure 8 shows the F-guide for the document of figure 1.

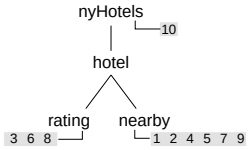

Figure 8: Function call guide

Since F-guides are trees, they can naturally be represented as XML documents, and therefore be serialized and queried just as the data they summarize. Moreover, this compact representation can easily be used to "approximate" the set of relevant function calls for a query. Indeed, it is easy to show that the linear path queries of Section 3 yield the same result on a document and on its F-guide. So rather than running the queries on the data one can can get better performance on its F-guide, which is typically much more compact.

Type-based filtering. The above queries give an over estimate of the relevant calls, which can be refined using types: We can discard the functions whose output type does not satisfy the query subtree rooted at the end of the linear path by which they were retrieved.

NFQ filtering. One can further filter the function call candidates using the NFQs. More precisely, for an NFQ q_v whose linear part is q_v^{lin}, the remaining query to evaluate checks for the conditions in q_v that don't appear in q_v^{lin}. This can be expressed as a relative XPath query, starting from the set of function calls returned by q_v^{lin}. For instance, for the NFQ of Figure 6(b), the remaining query is `.[..[*()⊕rating][*()⊕name[*()⊕"Best Western"]]]`.

To conclude, note that the F-guide can be built in linear time in the size of the document, by a single (document-order) traversal. The F-guide must also be maintained as the document evolves. This maintenance must be performed if the document is updated but also during query evaluation, when functions are invoked. The techniques resemble that of dataguide maintenance. Details are omitted.

7. PUSHING QUERIES

In general the result of a call may contain *more* data than is actually needed for evaluating the query. In such cases, to reduce data transfer and perhaps computation, it may be preferable to ask the function provider to send not the whole function result, but only the part useful for evaluating q. This amounts to *pushing* a query to the data source. For instance, in our running example, rather than invoking `getNearbyRestos` and getting all the near by restaurants, we would like to also ship the subquery

```
//restaurant[rating="*****",name=X,address=Y]
```

(with `X` and `Y` marked as result nodes) with the call. Thus, instead of all vicinity restaurants, only the name and address of five-stars ones are returned.

Pushing queries is a fairly standard technique in mediator systems [23], and involves issues such as verifying that the remote source is capable of evaluating them [24]. We focus here on issues that are particular to our setting, namely how to determine which subquery needs to be pushed, and how to use the returned results.

Which subquery to push over a function call. As previously explained, function calls are invoked when running the NFQA algorithm on a given layer of NFQs. We obtained f necessarily as the result of the NFQ q_v of some node v in q. The subquery to push over f is exactly the subtree rooted at v in the query, which we denoted sub_v^q in Section 5.

How to evaluate queries with pushed subqueries. When functions are invoked, the call parameters and the query to be pushed are sent to the function provider. Rather than returning the full answer, the latter returns bindings to the query variables that are result nodes. For instance, when pushing `//restaurant[rating="*****",name=X,address=Y]` over the call to `getNearbyRestos`, with `X` and `Y` marked as result nodes, the output consists of `X,Y` binding pairs, e.g.:

```
<tuple> <x>In Delis</x> <y>2nd Ave.</y> </tuple>
<tuple> <x>The Capital</x> <y>2nd Ave.</y> </tuple>
```

and not of `restaurant` elements. Subsequent queries that use

this data (the NFQs to be evaluated in the future, and q) need to be modified to take these binding into consideration.

There are two basic mechanisms for doing this. A first method is to actually *insert the result in the document* (replacing as before the function call) and modify the subsequent queries as follows. Let v be the query node for which f was discovered, and let $Z_1, \ldots, Z_k$ be the return nodes in sub_v^q. Then, in each subsequent query, replace sub_v^q by:
$sub_v^q \oplus$ `tuple[`z_1`=`Z_1`,...`z_k`=`Z_k`]`

This approach is very simple and follows naturally from the AXML model of query invocation. However, it has the drawback of leaving extraneous content in the document after evaluating q; `<tuple>` elements need to be eliminated afterwards. Alternatively, results from the service calls can be stored in an intermediary (temporary) storage, with the subsequent queries (the NFQs and the original query) modified to access them via *join*. This latter approach is indeed the one used in our implementation.

8. IMPLEMENTATION & EXPERIMENTS

All the ideas presented here have been implemented and tested in the context of the ActiveXML system [1], which provides persistent storage for AXML documents and allows users to declaratively specify Web services as queries over such documents. We used the algorithms described here to implement a module that decides which services need to be invoked for query evaluation, and pushes, when possible, the relevant subqueries to them. The system uses an XQuery-like query processor. The invoked queries are first *approximated* (in the safe sense of Section 6.1) by extracting their core tree patterns queries, and then fed to the above module. The signatures of functions are given in their WSDL specification, as an (A)XML Schema, which we also approximate, as explained in Section 6.1 by a graph schema. This model is employed for processing user queries as well as Web services defined by queries. Therefore, during the analysis of one query on one document, the same analysis may be applied, recursively, for the services invoked during the rewriting.

We ran several experiments to validate our techniques and briefly describe below a representative subset of them.

8.1 Experimental setting

Hardware and software environment. As a Web service communication infrastructure, the system uses the SOAP implementation of the Apache Axis platform. All the implementation (including the XML query processor) is done in Java. We performed the measures on a Compaq Evo N410c, with 256 MBytes of RAM and a PIII 1.2 GHz CPU, running Mandrake Linux 9.2.

Documents and functions used. The documents we use have a structure very similar to the sample from Figure 1. They all consist of `hotel` elements, which may include calls to the functions: `getRating`, `getNearbyRestos`, and `getNearbyMuseums`; we also use an extra `getNearbyHotels` function whose result is a set of hotels. The latter may contain similar calls, thus requiring a recursive relevance analysis.

We used the ToXgene [5] XML generator to produce documents. We fixed the document size to be 1.6MB [5], and varied the number of function calls in the document by instructing ToXgene to set the content of, e.g., `rating` elements f% of the time as a call to `getRating`, and (100-f)% of the time as materialized data. The same fraction f controls the probability of a `nearby` element to contain a call to `getNearbyHotel/Restos/Museums`, respectively, instead of plain data without function calls. The following table summarizes the parameters of the generated documents:

Value of f (%)	2	20	27	35
Number of function calls	466	757	970	1346

Queries. We report experiments performed with two queries:

`hotels//hotel`	(LIN)

`hotels/hotel[rating=‘‘***’’]`
`/nearby/hotel[rating=‘‘***’’]`
`/nearby/hotel[rating=’’***’’]`
	(TRE)

We chose these queries to illustrate a variety of scenarios. The linear query `LIN` is quite simple, but it searches at any depth in the document. `TRE` is a tree query, with branch tests at all levels, following only precise paths.

8.2 Experimental results

The results of our experiments are shown in Figure 9. In all of them, on the x axis we vary the number of service calls, by moving from one document to another. We consider four different filtering scenarios, using: only an F-guide; the F-guide and the type analysis; the F-guide and NFQ filtering; and finally, all three filtering techniques. The results also apply for a setting without F-guides, as our time measures here include the F-guide construction time. We used two approximations: NFQs expressed in XPath, thus with no value joins; and relaxed schemas as in Section 6. For our examples, these approximations happen to be precise.

Number of triggered service calls. At the top of Figure 9, we counted the calls triggered by the evaluations of `LIN` (left) and `TRE` (right). In the case of `LIN`, F-guide filtering doesn't eliminate any call, since the query traverses the whole document: thus, all function calls are triggered. Type-driven filtering, on the other hand, achieves a significant reduction, since it avoids the useless invocations of `getNearbyMuseums`, `getNearbyRstos` and `getRating`. The NFQ technique does not alter the impact of either F-guide, or type filtering, since `LIN` has no branch conditions.

For `TRE`, F-guide filtering leads to ignoring the calls found in regions of the document not traversed by the query; still, all calls under `hotel` elements, whether they bring ratings, hotels, museums, or restaurants are triggered. NFQ exploits the predicates on rating to avoid triggering such calls for hotels having the wrong rating. Type filtering eliminates irrelevant calls to `getNearbyMuseums` and `getNearbyRestos`. The cumulated effect of the three techniques leads to the biggest reduction, which is quite impressive: at far right, only 8 out of 1346 function calls are triggered.

Analysis time. At the bottom left of Figure 9, we measured the total time spent doing relevance analysis for `TRE`, that is, building the F-guide and analyzing it, testing function signatures, and evaluating NFQs. F-guide construction and analysis is very fast (the bottom curve); F-guide and type filtering is slightly more expensive. Running NFQs takes some time, therefore the curve for all three filtering techniques is higher. Note however that the three bottom curves vary very little with the number of service calls. The slight W-shape of the three-filter curve is due to a data variation in the XML documents that we were not able to control – the co-occurrence of three-star hotels at three successive

[5]The actual sizes of ToXgene documents vary slightly.

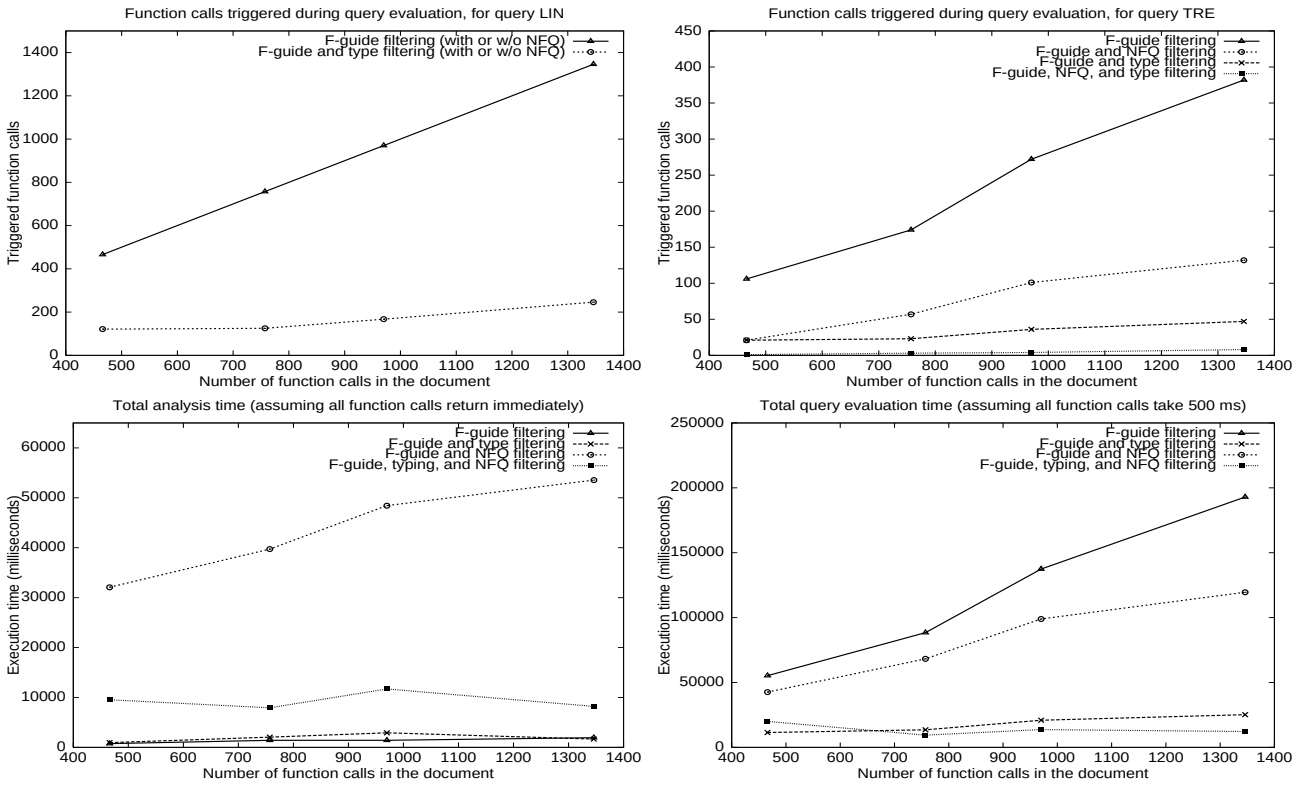

Figure 9: The effect of function call filtering: triggered calls (top), analysis and evaluation time (bottom).

levels was higher in some documents than in others. Using F-guide and NFQ filtering, but without checking the types (top curve) is a very bad choice: it leads to evaluating a large amount of NFQs, for many irrelevant calls to `getNearbyRestos`, `getNearbyMuseums` etc. This curve grows linearly with the number of services in the document.

Total evaluation time. The graph at the bottom of Figure 9, at right, tells a different story. Here we measure the total evaluation time, namely analysis *plus* service invocations, to capture the tradeoff between the time spend on analysis vs. the time saved by avoiding useless invocations. We set the function call duration to 500ms. This is a very minimal value, since the times we measured for calling a no-effect service call (constructing an empty-content outgoing SOAP message, sending it to the same host, sending an empty answer back, unpacking the message) varied in average between 500ms and 2500ms. In practice, a service call is likely to last much longer. In this graph, F-guide analysis alone becomes extremely expensive, due to the large number of function calls it triggers. F-guide and NFQ filtering is somehow better, since NFQs are effective in pruning function calls for `TRE`; still, in the absence of type checking, it is very expensive. F-guide and type filtering drastically reduce the total query evaluation time; adding NFQ to the mix brings a further slight improvement. The lowest two curves cross, when the extra time spent by three-techniques analysis pays off the time saved in function calls.

Similar results for `LIN` are omitted for space reasons.

To summarize, NFQ filtering is useful for queries having branch conditions, and does not affect the others. Simple F-guide analysis, without considering the functions' return types, is a very bad choice, since it triggers many irrelevant calls. Finally, and most importantly, F-guide, type, and NFQ filtering, together, are extremely cost-effective: the analysis pays off very significantly, both in terms of function calls and total evaluation time. Combining all filtering techniques, we measured more than a 10-fold reduction in the total evaluation time of `TRE`; in a more realistic setting, where Web service calls often last more than half a second, the improvement is likely to be even more important.

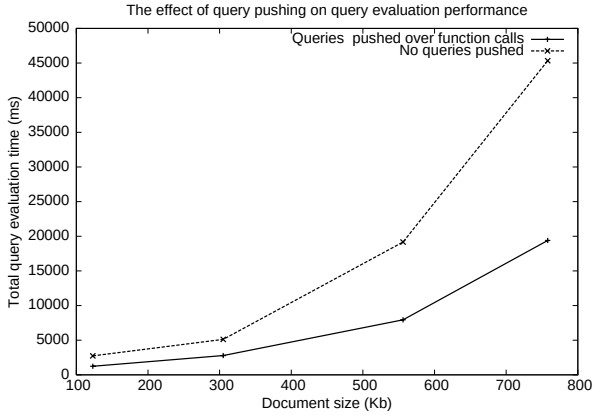

Figure 10: The effect of subquery pushing.

The effect of pushing queries. We also measured the effect of pushing subqueries over function calls. Figure 10 depicts the average total query evaluation time for queries in the style of `TRE`, with and without pushing. As expected,

data transfers are greatly reduced (up to three times on our fast LAN). The curves are quadratic due to the fact that the *getNearbyHotels* service is implemented by a query over the hotels document. The more hotels the document contains, the more nearby hotels there are, and the query will run on a document of size quadratic in the initial number of hotels.

9. CONCLUSION & RELATED WORK

This work is concerned with XML documents with embedded calls to Web services. Such documents are used in different systems [25, 17, 18, 1]. In the line of [1], we referred to them as AXML documents. We presented an algorithm to efficiently find and invoke the service calls relevant to a particular query, taking into account the output types of services, and pushing queries to them when possible. We also introduced an auxiliary access structure, F-guide, to speed up processing and presented results of experimental studies.

The evaluation of queries over AXML documents with lazy service calls is also the topic of the unpublished work of [4]. Their focus is on minimizing data materialization at a global level (using a generalization of Query-Subquery [27]) for systems with many interrelated documents in a peer-to-peer setting. Also set in the context of AXML, a previous work [3] considered the problem of distributing and replicating AXML data and services in a peer-to-peer setting. In contrast to both, we address here the specific problem of optimizing the evaluation of queries over *one* local AXML document. So, our work is clearly complementary to both, and indispensable for them to obtain good performances.

AXML documents may be seen as a means of integrating data sources, that combines mediation [23, 19] and warehousing [13]. There are indeed analogies between finding relevant calls and pushing queries to them, as proposed here, and query rewriting using views [14]. A novelty in our approach is that mappings between data sources are captured by service calls embedded in the data, with new relationships discovered at run-time, in the answers of service calls. This approach, based on ad-hoc dynamic discovery of mappings, is in the spirit of peer-to-peer data integration systems [15]. However, in systems like [15] the integration happens at the schema level, while in AXML it is at the data level. This strongly suggests combining the two approaches.

Our F-guides are inspired by dataguides [11]. Although they serve different purposes, they are both based on paths in the original documents. This is also in the spirit of adaptive path indices [9]. Invoking calls only when needed is a technique classically used in programming languages, under a variety of names, e.g., lazy reduction for lambda calculus [6]. The originality of our work comes from the particular setting, XML with embedded service calls.

We already mentioned various techniques that we believe are candidates to meet ours towards solving the general problem of query processing for AXML. It would also be interesting to consider techniques for minimizing communication, e.g., semi-join-based techniques. One should also consider an important classical aspect in mediator systems, namely source capabilities. This is also an issue in our context, for pushing queries to sources, and descriptions such as [24] could be used. In the spirit of peer-to-peer, one should also consider the issue of publication and discovery of data sources, see [26]. Issues such as quality of service, unreachability of some peers, substituting one data source for another should also be considered.

10. REFERENCES

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and web services integration (demo). In *Proc. of VLDB*, 2002.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *Proc. of ACM PODS*, 2004.

[3] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of ACM SIGMOD*, 2003.

[4] S. Abiteboul and T. Milo. Web Services meet Datalog. Technical report, INRIA, 2004.

[5] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for XML. In *Proc. of ACM SIGMOD*, 2002.

[6] H. P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 321–363. Elsevier, Amsterdam, 1990.

[7] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In *Proc. of ICDE*, 2003.

[8] P. Buneman, S. B. Davidson, M. F. Fernandez, and D Suciu. Adding structure to unstructured data. In *Proc. of ICDT*, 1997.

[9] C.-W. Chung, J.-K. Min, and K. Shim. APEX: an adaptive path index for XML. In *Proc. of ACM SIGMOD*, 2002.

[10] H. V. Jagadish et al. TIMBER: A native XML database. *The VLDB Journal*, 11(4), 2002.

[11] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, 1997.

[12] G. Gottlob and C. Koch. Efficient algorithms for processing XPath queries. In *Proc. of VLDB*, 2002.

[13] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. of ICDT*, pages 98–112, 1997.

[14] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[15] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proc. of the Int. WWW Conf.*, 2003.

[16] J. E. Hopcroft and J. D. Ullman. *Intro. to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[17] Jelly: Executable XML. `http://jakarta.apache.org/commons/sandbox/jelly`.

[18] Macromedia Coldfusion MX. `http://www.macromedia.com/software/coldfusion/`.

[19] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries over heterogeneous data sources. In *Proc. of VLDB*, 2001.

[20] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. of ACM PODS*, 2002.

[21] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. Dang Ngoc. Exchanging intensional XML data. In *Proc. of ACM SIGMOD*, 2003.

[22] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proc. of ACM PODS*, 1999.

[23] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proc. of ACM SIGMOD*, 1999.

[24] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. Query set specification language (QSSL). In *Proc. of WebDB*, 2003.

[25] J. Powell and T. Maxwell. Integrating Office XP Smart Tags with the Microsoft .NET Platform. `http://msdn.microsoft.com`.

[26] Universal Description, Discovery, and Integration of Business for the Web (UDDI). `http://www.uddi.org`.

[27] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. 1st Int. Conf. on Expert Database Systems*, 1986.

[28] The World Wide Web Consortium. `http://www.w3.org/`.