Tel Aviv University

The Raymond and Beverly Sackler

Faculty of Exact Sciences

Optimizing Query Evaluation with Distributed Query-Sub-Query

Thesis submitted in partial fulfillment

of graduate requirements for the degree

"Master of Sciences" in Tel Aviv University

School of Computer Science

By

Noam Pettel

Prepared under the supervision of Prof. Tova Milo

April 2006

Acknowledgements

I would like to express my deep gratitude to my supervisor, Prof. Tova Milo, for her guidance throughout this work, for her productive insights and suggestions and for her constant support and encouragement.

I thank Bogdan Cautis and Gabriel Vasile from INRIA for their invaluable help with Active XML.

To my lab-mates in Tel Aviv university I thank for their help, encouragement and the good atmosphere in the lab.

Lastly, a special thanks to my wife Perah for her belief in me, and to my son Ofir for giving me the strength.

Abstract

Research on deductive databases and Datalog led to beautiful results in the late 80s, but with little industrial impact, essentially because recursive data that required such technology was in practice rare. Years later, with networks everywhere, recursive data management is essential, and consequently this research is becoming relevant. Things are of course more complex than in classical Datalog: the architecture is often based on distributed autonomous peers and the interaction is often asynchronous. Nevertheless, one encounters again the management of a mix of intensional and extensional information in a recursive setting, and the old optimization methods proposed for Datalog meet the new needs.

Specifically, recent research showed that it is possible to adapt *Query-Sub-Query* (QSQ), a technique that optimizes the evaluation of Datalog queries, to a distributed setting, and use it to make query evaluations over distributed data more efficient. The problem was modeled using a distributed version of Datalog, which is called *dDatalog*, and optimized with a distributed version of QSQ, called *dQSQ*.

In this work we checked the feasibility of implementing a real system for optimization of distributed Datalog queries based on the dQSQ technique, examined the challenges that arise from it, and eventually built such a system that can be used in practice.

Our implementation of dQSQ is based on *Active XML* (AXML), a system with XML documents where some parts of the data are given explicitly, while other parts are given intensionally as calls to Web services. We defined a mechanism to represent dDatalog programs using AXML. We also defined a termination detection mechanism and showed how to embed it to the system. Finally, we developed a full-fledged dQSQ system, based on these mechanisms.

Contents

List of Figures

1 Introduction

Research on deductive databases and Datalog, a hot topic in the late 80s, led to beautiful results, with little industrial impact, essentially because recursive data that required such technology was in practice rare. Years later, with networks everywhere, recursive data management is essential. For instance, telecommunication systems interact with each other to gather routing data, possibly recursively. Also, a Web portal may want to retrieve information from some Web servers, referring to other servers recursively. In both cases, recursive data management is an essential aspect of the problem. Things are of course more complex than in classical Datalog: the architecture is often based on distributed autonomous peers and the interaction is often asynchronous. Nevertheless, one encounters again the management of a mix of intensional and extensional information in a recursive setting, and the old optimization methods proposed for Datalog meet the new needs.

A first attempt to "adopt" technologies from Datalog in favor of query evaluation over the Web, was made in [2]. It was shown there that it is possible to use optimizations from the Datalog world to make query evaluations over distributed data more efficient. In particular, one can adapt *Query-Sub-Query (QSQ)* [32, 6], an old technique from deductive databases, to a setting where autonomous and distributed peers share large volumes of interrelated data. QSQ is an optimization technique for query evaluation in deductive databases, aiming at minimizing the quantity of data that is materialized. The core of QSQ consists in the rewriting of a Datalog program given a query. In [2] QSQ was enriched to handle distribution, such that each peer can perform its own rewriting

with only local information.

Moreover, [2] demonstrates how to apply this optimization on a very practical application from the telecommunication field - the *diagnosis of asynchronous discrete event systems*. Although on the face of it, this does not appear to be a pure database application, the paper shows that (i) the problem can be suitably modeled using Datalog programs, and (ii) it can benefit from the large battery of optimization techniques developed for Datalog. The problem is modeled using a distributed version of Datalog, which is called *dDatalog*, and optimized with a distributed version of QSQ, called *dQSQ*.

Although the above idea is interesting, the distance between its theoretical development to implementing a real system for optimization of distributed queries in practice, is quite far. The purpose of this thesis was to check the feasibility of implementing such a system, examine the challenges that arise from it, and eventually to build such a system that can be used in practice.
When one decides to build a system which implements the above idea, there are numerous strategic decisions to make, such as:

- *Which underlying technology to use*. Two prominent technologies that arise in the context of distributed data management are *XML* and *Web services*. XML [34], a self-describing semi-structured data model, is becoming the standard format for data exchange over the Web. Web services started providing an infrastructure for distributed computing at large, independently of any platform, system or programming language. Together, they provide a framework for distributed management of information over the Internet.

- *What algorithms to use*. In this context it is important to emphasize that since we are dealing with a Peer-To-Peer (P2P) environment, the algorithms that we use for our system need to be adjusted to a distributed setting.

We have examined the various options and decided to implement a dQSQ system which is based on *Active XML (AXML)* [7, 3]. AXML documents are XML documents where some parts of the data are given explicitly, while other parts are given intensionally as calls to Web services that generate data. The analogy between deductive databases and active documents is quite strong. Think of an element node in a document as a collection. This collection may be given extensionally or specified intensionally with a call to some Web service. Such nodes play the role of a Datalog predicate, while calls to Web services play the role of Datalog rules. The recursion comes in naturally: think of a Web service calling a second Web service that calls the first.

We have defined and implemented a mechanism that represents a rewritten dDatalog program using AXML. In the AXML setting, an intensional relation is simulated by an active document that is enriched while new facts are deduced, and a rule is simulated by a Web service that produces facts.

Another important aspect of our work was the definition and implementation of the *termination detection* algorithm. Since the state of the Datalog program is distributed, it is more complex to detect the termination of query computation than in classical Datalog. We adopted a standard termination detection algorithm for distributed computing, in the style of [16], for detecting that all peers are in idle mode and that the

final result for the query has been obtained. The main challenge in this context was to define a mechanism that embeds the termination detection algorithm in our AXML-based dQSQ system.

The main contributions of this work are:

- Building a full-fledged system that uses the dQSQ algorithm to optimize query evaluation in a distributed environment.
- Implementing a distributed algorithm that rewrites a dDatalog program to its optimized QSQ form. Our implementation fits nicely in the AXML framework, as it uses Web services to send messages between the peers.
- Defining a representation of (rewritten) dDatalog programs using AXML, and implementing a translation process to translate dDatalog programs to a set of AXML entities.
- Defining a termination detection algorithm, implementing it and providing a mechanism to embed it in our AXML-based system.
- Introducing an interactive and dynamic Web user interface, which allows the user to interact with the dQSQ system, and in particular to submit queries and to view their results as they are obtained online.
- Contribution to the AXML framework: During the implementation of the dQSQ system, we contributed significantly to the AXML open-source code. Our contribution includes bug fixes, performance improvements, system stabilization and implementation of new features. Regardless its essentiality to the functionality of the

dQSQ system, other users of AXML will surely benefit from our contribution as well.

Outline of this Thesis The rest of this thesis is organized as follows. Section 2 gives the theoretical background, which includes an overview of the Datalog language, the QSQ optimization and their extension to a distributed environment. In section 3 we describe the implementation using the AXML framework. The termination detection algorithm and implementation are presented in section 4. In section 5 we give a guided tour of the dQSQ system. Section 6 discusses related work, and section 7 gives our conclusions, and areas for future research.

2 Distributed Datalog & Query-Sub-Query

In this section we present the theoretical background which was the basis for our work. The goal of the dQSQ system is to optimize evaluation of Datalog queries in a Peer-To-Peer environment. We give the required background that is needed to understand the context of our problem, step by step. We begin in section 2.1 with an overview of Datalog and Query-Sub-Query in the centralized case. We discuss the Datalog language, the evaluation of Datalog queries and its optimization with the Query-Sub-Query technique. In Section 2.2 we extend the discussion to a distributed setting, presenting a distributed version of Query-Sub-Query that was first shown in [2]. Finally, in Section 2.3 we discuss our enhancements to [2], which are required for the implementation of such a dQSQ system in practice.

The presentation in this section is not formal. The formal definitions of Datalog and Query-Sub-Query can be found in [6], and the formal extension to a distributed setting is given in [2].

2.1 Centralized Environment

Before we discuss evaluation of distributed programs, we consider local ones. We start with a short overview of Datalog programs, and continue with query evaluation and its optimization with Query-Sub-Query.

2.1.1 Datalog

Datalog is a language designed for deductive databases. As opposed to a relational database, where all the data is stored explicitly, a deductive database consists of explicit data, as well as logical rules that deduce

new facts. A Datalog program is a set of relations and rules that describe a deductive database. A Datalog program that computes the `ancestor` relation in a family is described in Figure 1. This program will serve as a running example throughout this thesis paper.

$$\begin{array}{lllll} \underline{rule\ 1} & ancestor(x,y) & \text{:-} & parent(x,y) \\ \underline{rule\ 2} & ancestor(x,y) & \text{:-} & ancestor(x,z), parent(z,y) \end{array}$$

Figure 1: A Datalog program

Based on the definitions presented in [6], we say that the `parent` relation is given *extensionally* as facts, while the `ancestor` relation is given *intensionally* i.e., defined by the program. Observe that the second rule is recursive, as the `ancestor` relation occurs both in the head and body of the rule. A Datalog program defines the relations that occur in heads of rules based on other relations. The definition is recursive, so defined relations can also occur in bodies of rules.

2.1.2 Naive Query Evaluation

We now consider evaluation of Datalog queries. Continuing with our *Ancestor* example in Figure 1, the query `q(x):-ancestor("Joyce",x)` computes all the descendants of Joyce i.e., it computes the `ancestor` tuples having "Joyce" in their first column and projects out the first column. Repeating the ideas presented in [2], we will first consider naive query evaluation and then its optimization with Query-Sub-Query.

We think of a naive query evaluation as a continuous flow of tuples. The evaluation works as follows. It starts with the query relation and

activates it. When a relation is activated, it activates all rules defining it. When a rule is activated, it activates all relations occurring in its body. A rule that is activated continuously receives tuples from the relations in its body and produces tuples (by evaluating the corresponding query). The computation terminates when (1) no more new rule or relation may be activated and (2) no new fact may be derived by any of the activated rules.

2.1.3 Query-Sub-Query Optimization

For Datalog, two main, closely related, optimization techniques for query evaluation have been studied, Query-Sub-Query (QSQ) [32] and Magic Set [11] (among others), that both aim at minimizing the quantity of data that is materialized. QSQ is described in detail in [32, 6]. Here we give some intuition on how it works. The crux of the QSQ optimization technique is to minimize the number of tuples derived. This is achieved by a rewriting of the Datalog program according to the given query and based on the propagation of bindings. QSQ starts from the rule defining the query. It processes the body of the rule from left to right. For each atom in the rule, it creates a "supplementary relation" to keep the bindings of variables that can be used for this atom. It obtains a new subquery, namely a call to the relation of this atom with the binding provided by the supplementary relation. More precisely, the QSQ rewriting is based on *binding patterns* and *supplementary relations*.

Binding Patterns For each relation name, consider *adorned versions* of the relation based on the bindings of the variables: e.g., for the 2-ary relation $ancestor$ above, $ancestor^{bb}$, $ancestor^{bf}$, $ancestor^{fb}$ and $ancestor^{ff}$

represent, respectively, the case with both variables bound, only the first one bound, only the second, and none. The top down, left-to-right evaluation of the rules determines the propagation of bindings.

Supplementary relations For each adorned relation and each position in the body of an adorned rule, a *supplementary relation* is introduced to accumulate the bindings relevant to that position. The notation $sup_{i,j}$ is used to denote a supplementary relation for position j in rule i.

The QSQ rewriting of the *Ancestor* Datalog program in Figure 1 is given in Figure 2.

As stated in [2], the QSQ evaluation has nice properties. It computes the correct answer to the query and materializes only a *minimal* set of tuples. In a central environment it is also guaranteed to terminate. The termination detection becomes a much more complex task in the distributed case. Section 4 is devoted to this problem.

$\underline{Query}$	$q(x)$	:-	$ancestor^{bf}(\text{"}Joyce\text{"}, x)$
	$in\text{-}ancestor^{bf}(\text{"}Joyce\text{"})$	:-	
$\underline{Rule\ 1}$	$sup_{10}(x)$	:-	$in\text{-}ancestor^{bf}(x)$
	$sup_{11}(x, y)$	:-	$sup_{10}(x), parent(x, y)$
	$ancestor^{bf}(x, y)$	:-	$sup_{11}(x, y)$
$\underline{Rule2}$	$sup_{20}(x)$	:-	$in\text{-}ancestor^{bf}(x)$
	$sup_{21}(x, z)$	:-	$sup_{20}(x), ancestor^{bf}(x, z)$
	$sup_{22}(x, y)$	:-	$sup_{21}(x, z), parent(z, y)$
	$in\text{-}ancestor^{bf}(x)$	:-	$sup_{20}(x)$
	$ancestor^{bf}(x, y)$	:-	$sup_{22}(x, y)$

Figure 2: The QSQ rewriting of the Ancestor Datalog program

2.2 Distributed Environment

Let us now extend our discussion to a distributed setting. The adaptation of Datalog and QSQ to a distributed, Peer-To-Peer environment was presented in [2]. We summarize here the main ideas of this paper, which are relevant to our work.

2.2.1 Distributed Datalog

In distributed Datalog (dDatalog) the relations and rules are distributed among several peers. A peer that hosts relation `r` holds the tuples of `r` as well as the rules defining `r` i.e., the rules which relation `r` is in their head. A distributed version of our *Ancestor* Datalog program is given in Figure 3. The notation ***ancestor@P*** indicates that relation `ancestor` is hosted by peer `P`. Peer `P` therefore also hosts rules 1 and 2 which have the `ancestor` relation in their head.

$$
\begin{array}{llll}
\underline{P} & relation & ancestor & \\
 & \underline{rule\ 1} & ancestor@P(x,y) & \text{:- } parent@R(x,y) \\
 & \underline{rule\ 2} & ancestor@P(x,y) & \text{:- } ancestor@P(x,z), parent@R(z,y)
\end{array}
$$

$$
\begin{array}{lll}
\underline{R} & relation & parent
\end{array}
$$

Figure 3: A dDatalog program

2.2.2 Naive Distributed Query Evaluation

Before we discuss distributed Query-Sub-Query, let us first reconsider newly activated relations in our naive query evaluation. For local relations, the treatment is the same as in the central case. For remote re-

lations, a request has to be sent to the remote peer hosting the relation. Then tuples start being produced in various peers and exchanged. The system reaches a fixpoint when no new relation may be activated and no new fact derived at any peer. It is easy to see that the result is exactly as in the centralized case. One problem here is the detection of termination. Since the state of the Datalog program is distributed, it is more complex to detect termination than in classical Datalog. We adopted a standard termination detection algorithm for distributed computing, in the style of [16], for detecting that all peers are in idle mode and that the final result for the query has been obtained. The termination detection algorithm and implementation are described in section 4.

2.2.3 Distributed QSQ Rewriting

We now optimize the query evaluation with the dQSQ algorithm, which is a distributed version of QSQ. The dQSQ processing starts at the peer where the query is submitted. As in centralized QSQ, we start with the rule defining the query, and then in a top-down fashion, process the body of rules defining each encountered relation, from left to right. We made a slight change from the algorithm presented in [2], due to implementation concerns. In [2], when a remote relation is encountered during the rewriting of a rule, the peer delegates the processing of the remainder of the rule (from the remote relation name to the right end of the rule) to the remote peer in charge of that relation. In our implementation, when a peer rewrites a rule, it makes the full QSQ rewriting of that rule. The generated rules which have a remote relation in their head are sent to the remote peer in charge of that relation.

Example We will illustrate the dQSQ rewriting process with an example. Assume that the query $q@P(x)$:- $ancestor@P(\text{"}Joyce\text{"}, x)$ is submitted at peer `P` for our *Ancestor* dDatalog program in Figure 3.
The steps of the rewriting process are as follows:

1. Peer `P` starts the rewriting process by determining the adornment of the relation in the head of the query. Since the first variable is bound to the constant value "Joyce" and the second variable is free, `ancestor` gets the adornment *bf*.

2. Next, peer `P` rewrites all the rules that define `ancestor`, namely the rules which the `ancestor` relation is in their head. Both *rule 1* and *rule 2* match this definition.

3. We will demonstrate the rewriting of *rule 2*. First, peer `P` sets the adorned version of the rule by propagating the binding of the variables in it. Peer `P` actually needs to set the adornments for all the intensional relations in the rule. This is pretty straightforward in our case and the adorned version of the rule is
$ancestor^{bf}@P(x, y)$:-$ancestor^{bf}@P(x, z), parent@R(z, y)$.

4. The next step is for peer `P` to define the supplementary relation for each relation in the rule. These are $sup_{20}@P(x)$, $sup_{21}@P(x, z)$ and $sup_{22}@R(x, y)$. Note that the last supplementary relation, $sup_{22}@R(x, y)$, is defined as a remote relation hosted by peer `R`. This is because it matches to the remote relation `parent`. The rules that define the three supplementary relations above are given in Figure 4.

5. Next, all the generated rules which are remote are sent to the peers

in charge of them. In our case, the rule
$sup_{22}@R(x,y)$:-$sup_{21}@P(x,z), parent@R(z,y)$ is sent to peer R.

6. Peer R receives the rule and continues the rewriting process recursively for the intensional relations in the body of that rule. In our case the rule has no intensional relations in it, thus the rewriting is finished at that point.

$$
\begin{array}{lcl}
sup_{20}@P(x) & :- & in\text{-}ancestor^{bf}@P(x) \\
sup_{21}@P(x,z) & :- & sup_{20}@P(x), ancestor^{bf}@P(x,z) \\
sup_{22}@R(x,y) & :- & sup_{21}@P(x,z), parent@R(z,y)
\end{array}
$$

Figure 4: The QSQ rewriting of *rule2*

To conclude, Figure 5 gives the resulting dQSQ program after the full rewriting. Observe that it is almost identical to that of Figure 2, obtained in the local case. It was stated in [2] that dQSQ is as optimal as QSQ, namely it materializes the same minimal information. An important point is that in dQSQ the rewriting is performed locally at each peer without any global knowledge.

Another important remark is that the query computation and the generation of results, may start even before the dQSQ rewriting is complete. This property is especially important in the context of the Web, where the number of sites transitively involved in a computation may be too large to explore exhaustively. Our implementation indeed performs the dQSQ rewriting and the query computation in parallel. When a rule is generated during the rewriting, it is immediately activated and tries to produce new tuples, even before the rest of the rewriting is completed.

$$
\begin{array}{lll}
\underline{P} & q@P(x) & :- \quad ancestor^{bf}@P(\text{"}Joyce\text{"}, x) \\
& in\text{-}ancestor^{bf}@P(\text{"}Joyce\text{"}) & :- \\
& sup_{10}@P(x) & :- \quad in\text{-}ancestor^{bf}@P(x) \\
& ancestor^{bf}@P(x,y) & :- \quad sup_{11}@R(x,y) \\
& sup_{20}@P(x) & :- \quad in\text{-}ancestor^{bf}@P(x) \\
& sup_{21}@P(x,z) & :- \quad sup_{20}@P(x), ancestor^{bf}@P(x,z) \\
& in\text{-}ancestor^{bf}@P(x) & :- \quad sup_{20}@P(x) \\
& ancestor^{bf}@P(x,y) & :- \quad sup_{22}@R(x,y) \\
\\
\underline{R} & sup_{11}@R(x,y) & :- \quad sup_{10}@P(x), parent@R(x,y) \\
& sup_{22}@R(x,y) & :- \quad sup_{21}@P(x,z), parent@R(z,y)
\end{array}
$$

Figure 5: The full distributed QSQ rewriting

2.3 Towards dQSQ Implementation

2.3.1 What Is Missing to Make It Work in Practice

The extension of Datalog and QSQ to a distributed setting was introduced in [2]. We have based our implementation of the dQSQ system on the ideas presented in this paper. However, the paper did not provide details about two aspects, which are crucial when one wishes to build a real dQSQ system. The first is the implementation details. The paper *did* refer to the implementation by suggesting the connection between dQSQ and *Active XML* (AXML) [7, 3], a system based on exchange of XML documents with embedded service calls in a Peer-To-Peer environment. Indeed, such a connection seems to be reasonable. In the AXML setting, an intensional relation can be simulated by an active document that is enriched while new facts are deduced, and a rule is simulated by a Web

service that produces facts. However, it turned out that the representation of dDatalog programs with AXML is not trivial at all. Therefore, we have defined and implemented such a representation mechanism, filling the gap that was left by the paper.

Another contribution to the completeness of [2] was the definition of the *termination detection* algorithm. Again, the paper refers to the need of such a mechanism, when evaluating a query in a distributed environment, but with lack of details. We have defined and implemented the termination detection algorithm, as well as a mechanism to embed it in our AXML-based dQSQ system.

2.3.2 System Overview

In this section we describe the dQSQ system architecture. The distributed system is composed of four components, each running on every peer:

1. The distributed QSQ rewriting process that rewrites a dDatalog program to a new program in a QSQ format. The system receives as input a dDatalog program, distributed among several peers. When a query is submitted at one of the peers, this peer initiates the dQSQ rewriting process. At the end of the process each peer holds its part of the resulting dQSQ program.

2. A translation process that translates a dQSQ program to an implementation dependent format. Since we chose Active XML to be the platform for the dQSQ system, this process translates every rule and every relation that were generated by the dQSQ rewriting process, to a set of AXML entities, which are needed to run the dQSQ

program in the AXML environment.

3. The query evaluation process that computes the query result by activating the generated dQSQ rules.

4. The termination detection process that runs parallel to the query evaluation process and decides when the computation is finished i.e., when all peers are idle and the final result of the query is obtained.

It is important to emphasize that the four processes run in parallel. Apart from making the system more efficient, the parallelism here is a necessity since the system does not know when the distributed dQSQ rewriting process (first process above) ends. Thus, when a new rule is generated by the rewriting process, it has an immediate effect on the other three processes: It is immediately translated to Active XML (second process); it is activated to produce new facts (third process); and the termination detection mechanism (fourth process) is updated accordingly.

A typical scenario of the system's work is as follows. It receives as input a dDatalog program, distributed among several peers. The user then submits a query at one of the peers. The peer that receives the query initiates the dQSQ rewriting process. It starts with the rule defining the query, and then in a top-down fashion, processes the body of the rules defining each encountered *local* relation, from left to right. The generated QSQ rules which are remote (have a remote relation in their head) are sent to the remote peer in charge of that rule. The remote peer continues the rewriting process by recursively rewriting the local relations

that appear in the body of the rule it received. During the dQSQ rewriting process, every generated rule is immediately translated to an AXML format, activated and starts to produce new facts. The termination detection process samples the state of the system periodically and notifies the user when the query computation process is finished.

2.3.3 Overview of Termination Detection

Since the state of the Datalog program is distributed, and there is no single peer with the full picture of the state of the query computation, it is more complex to detect termination of the query computation than in classical Datalog. However, this is rather a standard problem in distributed computing. Following the principles presented in [2], we defined a mechanism in the style of [16]. We implemented a distributed termination detection algorithm that runs parallel to the dQSQ query computation and determines whether the computation is finished i.e., all the data that is needed to answer the query has been materialized at all the peers. The general idea is as follows. We assume that communications are asynchronous and that each message eventually arrives and acknowledged. At some point, the site that started the query decides to check for termination. It calls all the sites that it directly invoked and asks them whether they completed. These sites contact the sites they invoked, and so on. A difficulty is the management of cycles in the invocation graph of the query, that are the result of recursive calls. Intuitively, we need to decide termination using some spanning tree of this graph. A site answers positively if (i) it is idle (i.e., cannot produce more data), (ii) all the data it has sent has been acknowledged and (iii) all its successors believe the computation is terminated. The termination detection

algorithm and its implementation in AXML are described thoroughly in Section 4.

3 dQSQ in Practice

3.1 Introduction

dQSQ is a prototype system for evaluations of distributed Datalog queries, optimized with a distributed version of the Query-Sub-Query technique. dQSQ is available for download from the Web at `http://www.cs.tau.ac.il/~milo/DQSQ/dqsq.html`. The implementation uses the *Active XML* system as a platform for managing data exchange in a Peer-To-Peer environment. Therefore, installation of *Active XML* is a prerequisite. The full details about installing and running dQSQ can be found in the dQSQ User Guide, which is also available for download in the above Web address.

This chapter is organized as follows. Section 3.2 gives an overview of the Active XML framework. In Section 3.3 we describe how we used Active XML for dQSQ. Finally, Section 3.4 overviews the dQSQ software architecture.

3.2 Active XML ++

As already mentioned, we implemented the dQSQ system using the Active XML [7] framework. To give you the needed background, we summarize in this section the overview presented in [3]. The "++" in the title refers to our contribution to Active XML, which is described in Section 3.2.6 and in Appendix A. Later on, in section 3.3 we describe the implementation details.

3.2.1 AXML Overview

In this section we give a general overview of the Active XML framework, based on [3]. Active XML (AXML in short) is a framework for distributed data management that addresses the concerns arising from the evolution of the World Wide Web. Such concerns are (i) the high heterogeneity and autonomy of data sources, and (ii) the scale of the Web. The AXML approach is based on XML and Web services. *XML* [34] is a data model that raised a considerable interest among the data management community as a standard for data exchange between remote applications, notably over the Web. The publication of XML data and the access to it are facilitated by *Web services*, i.e., by programs taking XML parameters and returning XML results. The WSDL [33] and SOAP [31] standards enable describing and calling these remote programs seamlessly over the Internet. Peer-To-Peer (P2P) architectures for data integration over the Web propose an alternative to centralized architectures, and are already spreading, notably in the context of file-sharing. These architectures capture the autonomous nature of Web systems, and their ability to act both as producers of information (i.e. as servers) and as consumers of information produced by others (i.e. as clients).

AXML is a language that leverages Web services for distributed data management and is put to work in a P2P architecture. AXML documents are XML documents with embedded calls to Web services. Such documents are enriched by the results of invocations of the service calls they contain. The AXML model also defines AXML services; these are Web services that exchange AXML documents. AXML services lead to powerful data-oriented schemes for distributed computation, where sev-

eral systems dynamically collaborate to perform a specific data management task, possibly discovering new relevant data sources at run-time. AXML seamlessly combines *extensional* data (expressed in XML) and *intensional* data (the service calls, that provide means to get data). An AXML "peer" is a repository of AXML documents. It acts both as a client, invoking the Web services embedded in the documents, and as a server, providing Web services defined as queries over the active documents it contains. The approach gracefully combines stored information with data defined in an intensional manner as well as dynamic information.

3.2.2 AXML Documents

Active XML documents are based on the simple idea of embedding calls to Web services inside XML documents. More precisely, AXML defines an XML syntax to denote service calls, and allows elements conforming to this syntax to appear anywhere in a document. Intuitively, these calls represent some (XML) information that is not given extensionally, but *intensionally*, by providing means to get the corresponding data. Service calls can be *materialized*, which means that the associated Web service is invoked (using the SOAP protocol), and the results it returns enrich the document, at the location of the service call. Figure 6 is a simple AXML document, that represents a local newspaper homepage. Note that this document is a syntactically valid XML document, where service calls are the elements labeled `axml:sc`. This document consists (i) of some extensional information: the title and date of the newspaper, and a news story about Google, and (ii) of intensional information: a service call to get the weather forecast, and another one to get a list of current exhibits

from a local guide. For each service call, the Web service that needs to be called is identified by the service attribute of the `axml:sc` element. For instance, `getEvents@TimeOut.com` means that we want to invoke the service `getEvents` provided by the `TimeOut.com` server. This is actually a simplified syntax, used here for the sake of clarity. The full syntax accounts for all the information needed to invoke the service using the SOAP protocol.

The sub elements below service calls are the parameters of the calls. For instance, the call to `forecast@weather.com` uses the name of the city `Paris` and a unit as parameters. In general, parameters can be arbitrary XML data, and they may even contain service calls themselves. The values of the parameters to the calls in Figure 6 are given explicitly. One can also use an XPath [36] expression to point to the value of the parameter, which is located elsewhere in the document.

3.2.3 AXML Services

As mentioned earlier, AXML documents are syntactically valid XML documents, and therefore can be parsed and processed by any XML-aware system. Moreover, just by following the SOAP and WSDL specifications, any Web services-aware system is able to materialize the calls embedded in AXML documents. Thus, AXML documents can be universally understood, and therefore can be exchanged between systems. This observation naturally leads to introducing AXML services, i.e., Web services accepting AXML documents as input parameters, and returning AXML documents as results. The signature of such services may be specified in WSDL. Observe that when AXML services are involved, materialization becomes a recursive process, since calling an AXML

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://purl.org/net/axml">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <edition>Paris</edition>

  <weather>
   % service call
    < axml:sc service="forecast@weather.com" >
      <city>Paris</city>
      <unit>Celsius</unit>
    < /axml:sc>
  </weather>

  <exhibits>
   % service call
    < axml:sc service="getEvents@TimeOut.com">
      exhibits
    < /axml:sc>
  </exhibits>

  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</rss:title>
      <rss:pubDate>
        Feb 18, 2003 10:36:03 GMT
      </rss:pubDate>
      <rss:description>
        Google just acquired Pyra labs,
        the company that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 6: An Active XML document

service may return some data that may contain new service calls that also need to be materialized, and so on recursively. This is illustrated in Figure 7, which gives the state of our sample document after the invocation of the `getEvents@TimeOut.com` service. This service answered with a list of exhibits, plus a service call to get more exhibits from `Yahoo`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<newspaper xmlns="http://lemonde.fr"
           xmlns:rss="http://purl.org/rss"
           xmlns:axml="http://purl.org/net/axml">
  <title>Le Monde</title>
  <date>2-Apr-2003</date>
  <edition>Paris</edition>

  <weather>
    <temp>16</temp>
  </weather>

  <exhibits>
    <exhibit>
      <name>Naive Painting in Ancient Greece</name>
      <location>Le Louvre</location>
      <from>25-Apr-2003</from>
      <to>25-May-2003</to>
    </exhibit>
    ...
    < axml:sc service="getExhibits@Yahoo.com">
      <city>Paris</city>
    < /axml:sc>
  </exhibits>

  <stories>
    <rss:item id="cx_ah_0_218">
      <rss:title>Google goes Blog-Crazy</rss:title>
      <rss:pubDate>
        Feb 18, 2003 10:36:03 GMT
      </rss:pubDate>
      <rss:description>
        Google just acquired Pyra labs,
        the company that makes Blogger.
      </rss:description>
    </rss:item>
    ...
  </stories>
</newspaper>
```

Figure 7: After invoking `getEvents@TimeOut.com`

Pull: Query and Update Services The basic way to define a service in an AXML peer is to write a parameterized query over its repository of AXML documents. Figure 8 contains a sample service definition.

```
let service GetExhibitsByLocation($loc) be
for $a in document("newspaper.xml")/newspaper/exhibits,
    $b in $a//exhibit
where $b@name=$loc
return <exhibits> $b </exhibits>
```

Figure 8: An AXML Service definition given as a query

The queries in the AXML system are written in the X-OQL language [8], an XQuery-like query language, but to simplify the presentation we use here the syntax of XQuery [37], which is more commonly used and is currently becoming the standard language for querying XML documents. The query in Figure 8 uses the `newspaper.xml` document from the repository, that may be similar to the one of Figure 6. When a call to this service is received by the peer, the `$loc` variable is bound to the transmitted parameter value (the location), then the query is evaluated, and its result is returned as an answer. It should be noted that any kind of AXML fragment can be passed as a parameter to the service (e.g., a service call, a forest of AXML trees, or an atomic value). Note that the query answer may naturally contain service calls, since the query is evaluated on the AXML parameters and the AXML documents of the repository.

Service call attributes: A service call element provides information on how and when to activate it through attributes. Two of the important attributes are *followedBy* and *frequency*.

The *followedBy* attribute allows to chain evaluation of service calls

inside the same AXML document. The value of this attribute is an XPath expression pointing to a service call element. For example,
`followedBy="//axml:sc[@service='getMovies']"`
specifies that right after the activation of the current service call has completed, the `getMovies` service call should be activated.

The *frequency* attribute defines when the service call will be activated. It can have one of the following values:

- once: The service call will be activated only once at start-up time.
- lazy: The service call will be activated only when needed, e.g., when its result may be useful to the evaluation of an X-OQL query, or when this service call is a part of a chain evaluation of service calls, defined with the *followedBy* attribute.
- on date: Specifies when exactly the service call will be activated.
- every X: Means that the service call will be activated every X milliseconds.

AXML also supports update services, that have side effects on the repository documents. These are also defined declaratively, as an extension to X-OQL.

Push: Continuous Services A server may work in pull mode, as described above, but may also work in push mode: The client subscribes to a particular push service, by issuing a service call, then the server asynchronously sends a stream of messages as an answer.

These services may also be specified declaratively, e.g., using parameterized queries. In the AXML peer implementation, any pull service

can serve as a basis for a continuous service. The latter will construct its answers by periodically reevaluating the wrapped pull service. A number of additional parameters can be specified, to control how a continuous service operates:

- A frequency of the messages (e.g., daily), which indicates to the service how much time to wait before reevaluating the service call and sending the updated result to the client.
- Some limitations, such as the duration of the subscription or some size limit on the data that is transmitted in each message.
- A choice of representation for the changes: send consecutive versions, send a delta or an edit script for the changes, (as in, e.g., [15]), send a notification and publish the changes.

Typically, an AXML peer that provides a continuous service supports/allows a limited subset of these choices among which the client chooses, by specifying the corresponding parameters in the subscription call.

The implementation of continuous services in the AXML peer emulates an asynchronous behavior on top of the widely used, synchronous HTTP protocol. Each AXML peer possesses a dedicated "callback" Web service whose purpose is to receive answers to any continuous services the peer has subscribed to. When a subscription call is issued, the client AXML peer automatically adds to the call some (intensional) information about its callback service. The peer that provides the continuous service then sends its answers asynchronously to that callback service, which integrates them at the right place (i.e., next to the corresponding subscription call).

Event-Sensitive Continuous Services The default behavior of a continuous service is to wakes up periodically (according to the `frequency` parameter that is specified by the client), reevaluate the service call and send the updated result to the client. We developed a new feature, which we call *Event-Sensitive Continuous Services*, and added it to the AXML framework. With this feature, a client who registers to a continuous service may ask from the server to reevaluate the service call only when an *event* occurs. The definition of events is application dependent and could be, for instance, the receiving of new data by the server. In the implementation of this feature, the application should call a Web service that produces the event. The details are given in Appendix A.

3.2.4 AXML Peer

To effectively support the AXML language, a distributed architecture based on the *Peer-To-Peer* paradigm, where each participant may act both as a client and as a server, was adopted. The participants of this architecture are called *AXML peers*. Their features and implementation are the topics of this section.

Since the AXML language is based on the use of Web services, the architecture relies on invocations of Web services for all communications between participating peers. AXML peers have essentially three facets:

- *Repository:* The first task of an AXML peer is to manage persistent information, very much like a traditional database management system. Since the goal is to support the AXML model, data will naturally consist of *persistent* AXML documents.

- *Client:* In order to take advantage of the intensional data present in its repository, an AXML peer may invoke the corresponding Web services. By doing so, it acts as a client of other peers. The AXML language is extended with features to control the activation of service calls.

- *Server:* An AXML peer may also provide Web services for other peers. These will typically be defined using queries or updates over the documents of its repository, and will indeed be AXML services.

The AXML architecture is illustrated in Figure 9, which shows the main internal components of an AXML peer. AXML peers exchange AXML data with each other through their Web services using the SOAP protocol. Existing Web service providers (such as Google) and clients (such as the Mozilla browser) naturally fit in this architecture, since all exchanges follow standards for Web services. However, the exchange of AXML documents will only occur between systems that understand the AXML language.

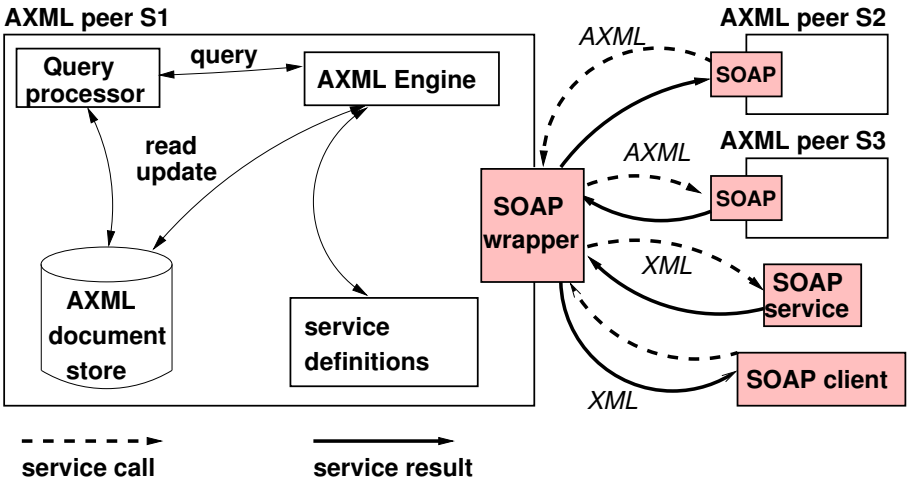


Figure 9: The AXML Peer-To-Peer architecture

3.2.5 AXML Implementation

The AXML approach mainly relies on the notion of AXML documents, and on standards for Web services. Therefore, any system that understands/implements these can join the AXML party. The AXML peer, whose main principles and features were presented above, is such a system. It was built to take fully advantage of the AXML model, by supporting both its client-side and server-side features and managing persistent AXML data. It was typically designed to run on a fixed desktop workstation, with a permanent network connection. The AXML system is distributed in OpenSource [9] within the ObjectWeb framework [27].

3.2.6 Our Contribution to AXML

During the implementation of the dQSQ system, we contributed significantly to the AXML open-source code. Our contribution includes bug fixes, performance improvements, stabilization and implementation of new features. Regardless its essentiality to the functioning of the dQSQ system, other users of AXML will surely benefit from our contribution. Theoretically, AXML should fit smoothly as the platform for the dQSQ system. Practically, we have encountered many problems during the implementation, some of which we fixed, but several problems remained as limitations. In general, when we started working with AXML our impression was that the system could not handle the load generated by the dQSQ system. We found a system that was not stable and not scalable, such that when many AXML services were running simultaneously, the system failed, reached an unstable state or performed poorly. This is especially problematic for dQSQ, since even a small dDatalog program

becomes big after being rewritten. Recall, for instance, the rewritten dDatalog program in Figure 5 which is the result of the program in Figure 3.

All our fixes and changes in AXML are described in Appendix A.

3.3 dQSQ Using Active XML

This section describes our representation of the rewritten dDatalog program in AXML. The process that translates the program to AXML runs parallel to the dQSQ rewriting process. Whenever a new rule is generated by the rewriting process, it is sent to the translation process, and the rule, as well as the relations in it, are being translated to AXML as will be described next. The description in this section will not include the additions that had to be made due to the termination detection algorithm. The termination detection algorithm and implementation are discussed in section 4.

We will demonstrate this representation using our running example of the *Ancestor* program. Figure 5 contains the resulting dQSQ program for the query $q@P(x)$:- $ancestor@P(\text{"}Joyce\text{"}, x)$.

3.3.1 High-Level Overview

Since the full representation is complex, we will describe it step-by-step. We begin with a high-level overview of the representation that omits some of the technical details. In the next section we extend the representation until we reach its final form.

Relations A relation R(x_1,...,x_n) is represented by an AXML document `R.xml` that contains the tuples of the relation. If the relation is exten-

sional, the document is static, as the tuples exist in the document from the moment it is created, and the document is not changed during the query computation.[1] Figure 10 sketches the structure of the extensional relation $parent$ from our *Ancestor* example. For clarity, from now on we will describe the structure of XML documents as *trees* and not as XML text. This is a standard representation in which a parent-child relationship in the tree matches to an element-subelement in the XML document.

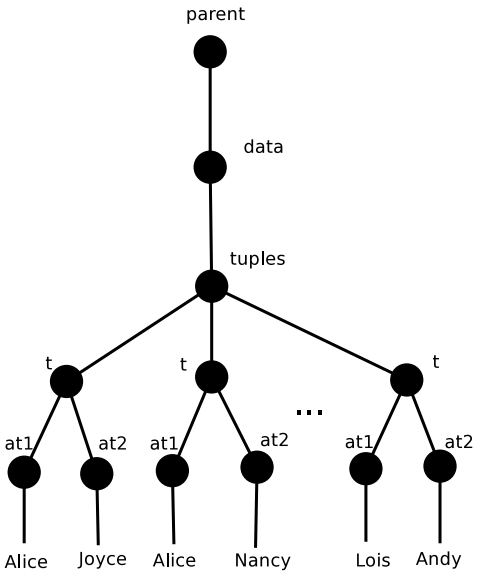


Figure 10: Representation of an extensional relation in AXML

If the relation is intensional, the document is dynamic as it contains service calls to add new tuples based on data in other (possibly remote) relations, while the query computation is in progress. Figure 11 sketches the structure of the intensional relation $ancestor^{bf}$. For every rule that defines the relation there is a corresponding service call element.

Figure 12 describes the $ancestor^{bf}$ relation after the service calls in it

[1] In AXML terms, a static document is a document with no service calls.

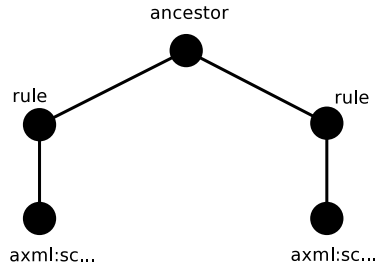


Figure 11: Representation of an intensional relation in AXML

were materialized. One of the service calls returned the tuple ("Joyce", "Lois"), denoting that "Joyce" is an ancestor of "Lois". The result of the service call is inserted next to the `axml:sc` element.

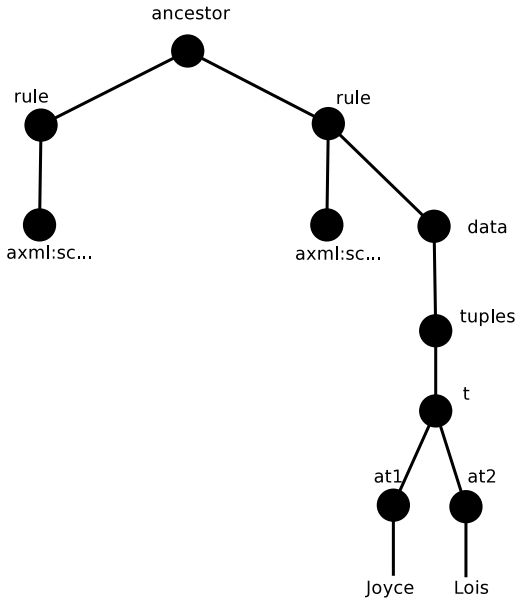


Figure 12: An intensional relation after data insertion

For every relation the system defines an X-OQL query service that returns data from the service's document according to a set of parameters that are specified by the caller. For example, the service *Service_parent*(x,y) is defined for the relation $parent$. The service can get

both bound and unbound parameter values, where an unbound parameter is denoted by the reserved character '*'. The call *Service_parent*("Alice",*), for instance, will return all the tuples of relation $parent$ having "Alice" in their first column. In other words it will return all Alice's children. Figure 13 contains a simplified definition of *Service_parent*. Although in the implementation we used the X-OQL language to define query services, we use here the syntax of XQuery for simplicity.

```
let service Service_Parent($x, $y) be
for $t in document("parent.xml")/parent/data/tuples,
where ($t/at1=$x or $x="*") and
      ($t/at2=$y or $y="*")
return $t
```

Figure 13: An AXML query service

As a matter of fact, we enable the caller of each service to pass an array of parameters. For instance, the client can specify that he wishes to get all the children of "Alice" and all the children of "Joyce" in the same call. In such a case the parameter passed to the service is {("Alice",*), ("Joyce",*)}. The parameter of the service is actually an AXML tree, and the elements of the array are represented as nodes in the tree.

We wrapped every service as a continuous service, such that new tuples are inserted asynchronously to the calling document when there is new data relevant to the request, while the query computation process evolves.

Rules The dQSQ rewriting algorithm creates rules that have at most two relations in their body. When a rule of the form $R(x_{i_1}, ..., x_{i_n})$:-$P(x_{j_1}, ..., x_{j_k}), Q(x_{h_1}, ..., x_{h_l})$ is activated, new tuples arrive at relation R based on join of the data in relations P and Q. In our

implementation this process is composed of three steps. Continuing with our running example, consider the rule $sup_{22}@R(x,y)$:-$sup_{21}@P(x,z), parent@R(z,y)$ that was generated by the dQSQ rewriting at peer `R`. The system at peer `R` defines a helper AXML document that contains service calls to receive the relevant data from the remote relation sup_{21} and from the local relation $parent$. The three steps are as follows:

1. Initially, the helper document contains a service call to get all the tuples of sup_{21}, namely the call *Service_*sup_{21}(*,*). This is illustrated in Figure 14. Each parameter of the service is denoted as a separated `param` element. Each `param` element has two sub elements: `key` that holds the name of the variable, and `value` that holds the value passed to the variable.

2. Based on the data received from sup_{21}, the system builds a service call to get data from $parent$. If, for example, sup_{21}, which is a binary relation, returned ("Alice","Joyce") as an answer, then the parameters of the call to *Service_parent* will be ("Joyce", *), as required by the definition of the rule. This is illustrated in Figure 15. Whenever sup_{21} returns new data, the system builds a new service call to *Service_parent* with the corresponding parameters.

3. The last step is to transfer the resulting data from the helper document to the relation in the head of the rule. For this purpose, sup_{22} calls a special service that pulls the relevant data from the helper document.

Figure 16 illustrates the process of deducing new facts for the rule $sup_{22}@R(x,y)$:-$sup_{21}@P(x,z), parent@R(z,y)$. In 16(a), the helper

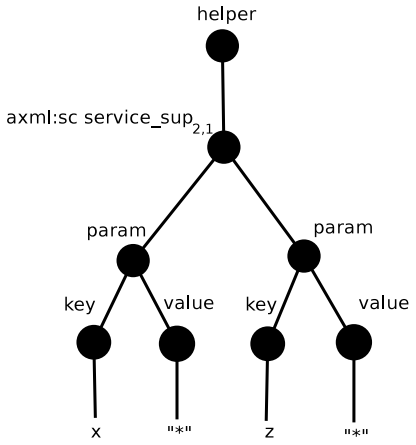


Figure 14: The helper document - step 1

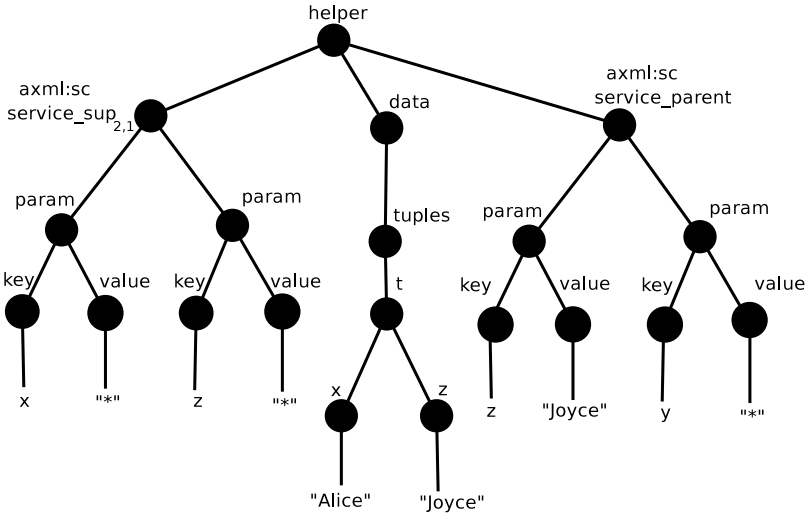


Figure 15: The helper document - step 2

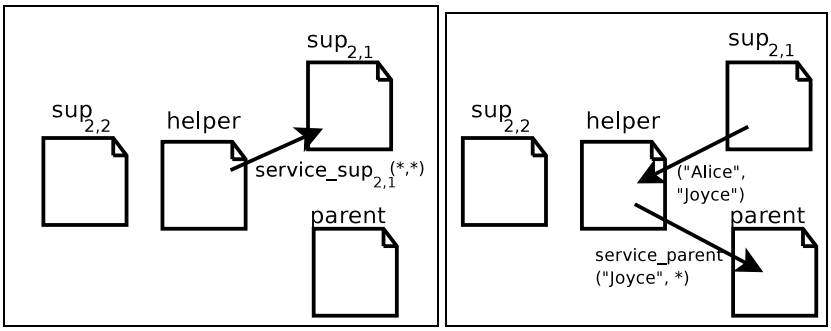
document registers on sup_{21} with (*,*). In 16(b), sup_{21} returns the reply ("Alice", "Joyce") and as a result, the helper document registers on $parent$ with ("Joyce", *). In 16(c), $parent$ returns the reply ("Joyce", "Lois"), and as a result the helper document sends the tuple ("Alice","Lois") to sup_{22}. This is the first tuple that was deduced by the rule. In 16(d), sup_{21} itself deduced ("Alice", "Ruth") as a new fact. Since the helper document registered on sup_{21} with (*,*), this tuple is relevant data for this registration. Therefore, sup_{21} sends it to the helper document. Consequently, the helper document makes a new registration on $parent$, this time with ("Ruth", *) as a parameter.

3.3.2 Detailed Description

The overview that was presented in the last section describes a relatively trivial representation. However, with such a simple structure of AXML documents and services, we could not reach the desired outcome. We will now extend the description and describe the full representation of programs using AXML.

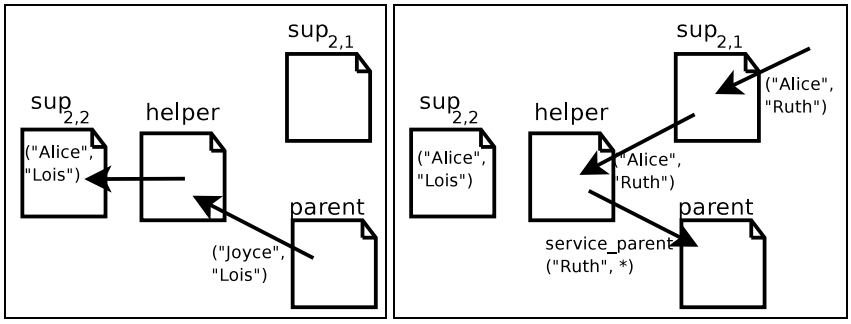
Sending incremental data By default, when a continuous service wakes up, it reevaluates the client's request and sends the updated result. We would like each service to send only new data to the client, namely incremental data. With AXML this goal is feasible. AXML allows the user to define a *diff service* for each continuous service. The *diff service* is a Web service that pre-processes the reply of the continuous service before it is sent to the client.
We use this mechanism as follows:

- For each continuous service we keep a database of all its registered



(a) Helper document registers on sup_{21}

(b) Helper document receives data from sup_{21} and registers on $parent$



(c) Helper document receives data from $parent$ and sends it to sup_{22}

(d) sup_{21} sends new data and helper document makes a new registration (with different parameters) on $parent$

Figure 16: Deducing new facts with rule $sup_{22}@R(x, y)$:- $sup_{21}@P(x, z)$, $parent@R(z, y)$

clients. For each client the service keeps all the data that it previously sent. The database that we used is a simple XML file, whose structure is illustrated in Figure 17.

- We implemented a *register* Web service , which registers the new client in the database. Before a client calls a continuous service, it calls the *register* service. To implement this, before every service call to a continuous service, the translator adds a service call to the *register* service. Figure 18 adds the call to *register* to the helper document in Figure 14. The broken arrow with the "followedBy" caption between the two service calls represents the *followedBy* attribute that tells AXML to run two services sequentially[2]. We will use this notation throughout this paper. To use the *followedBy* mechanism, we add to the first service call the *followedBy* attribute, with its value being the name of the second service; the frequency of the second service is assigned with the value "lazy", which tells AXML to wait with its execution until there is a specific demand to run it. In our example, the *register* service is called first, and only after its execution is finished, the *followedBy* attribute tells AXML to call $service\_sup_{21}$.

  Observe that this mechanism of calling *register* before calling a continuous service, was applied on *every* call to a continuous service in the system. From no on, however, we will omit the call to *register* in our diagrams, for the sake of simplicity.

- We implemented a *diff service* that compares the reply to the data that was previously sent to this client, and sends only new data.

[2]The *followedBy* attribute was presented in Section 3.2.3, when we discussed service call attributes in AXML

If there is no new data, the service does not return any reply to the client. In this fashion, the client receives a reply only when it contains new data.

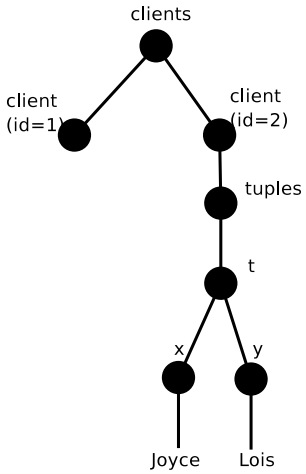


Figure 17: The clients database of a continuous service

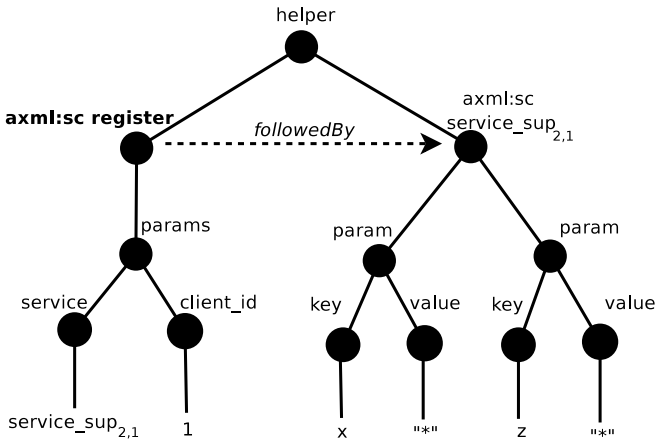


Figure 18: Adding the *register* service call

Translating rules Let us now describe the rule translation process with more details. We will demonstrate the translation on the rule $r1 : sup_{22}@R(x, y)$:-$sup_{21}@P(x, z), parent@R(z, y)$.

$r1$ is the unique ID of the rule. As mentioned in the high-level overview, the system creates a helper AXML document at peer R that contains service calls to receive the relevant data from the remote relation sup_{21} and from the local relation $parent$. Initially the helper document contains a service call to get all the tuples of sup_{21}, namely *Service_*sup_{21} is called with (*,*). Based on the data that will be received from sup_{21}, the system needs to build a service call to get data from $parent$. To achieve this we use the following mechanism:

- The system creates a *buildQuery* service that builds the parameter value for *Service_parent* based on the reply that was received from sup_{21}. Observe that every rule has its own *buildQuery* service.
- The service call to *Service_*sup_{21} contains additional two parameters: (1) the name of the *buildQuery* service, namely *buildQuery_r1* and (2) the name of the next service to be called, namely *Service_parent*.
- *Service_*sup_{21} returns, together with the data, two service calls: (1) to *buildQuery_r1* service and (2) to *Service_parent*.
- The two service calls are executed sequentially using the AXML *followedBy* attribute. First, the *buildQuery* service is called. The parameter passed to the service is the data returned from sup_{21}. AXML enables to point to parameter values using the XPath language. (See Section 3.2.2 in our discussion about parameters to service calls). We use it here to pass the result of *Service_*sup_{21} as the parameter to *buildQuery*. Next, *Service_parent* is invoked. The reply returned by *buildQuery* is actually the parameter value for *Service_parent*. Here too, we use XPath to point to the result of

one service call as the parameter value of another service call.

Note that this mechanism generates a new service call with different parameters to *Service_parent* with every reply that is received from *Service_*sup_{21}. The state of the extended structure of the helper document, as was described above, is illustrated in Figures 19, 20 and 21. Figure 19 is the document in its initial state, which contains the two additional parameters - *buildQueryService* and *nextService*; Figure 20 is after getting the results from *Service_*sup_{21}; and Figure 21 is after getting the results from the *buildQuery* service. The curved arrows with the "xpath" caption denote the usage of XPath as described above.

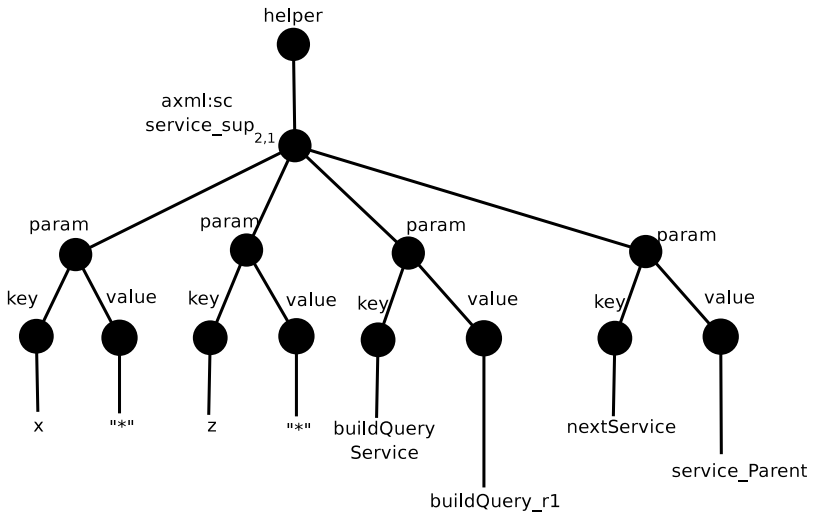


Figure 19: Helper document in its initial state

The next step is to transfer the resulting data from the helper document to the relation in the head of the rule, meaning to sup_{22}. For this purpose, we use the following mechanism:

- A special service called *Rule_r1*, that pulls the relevant data from the helper document is created.

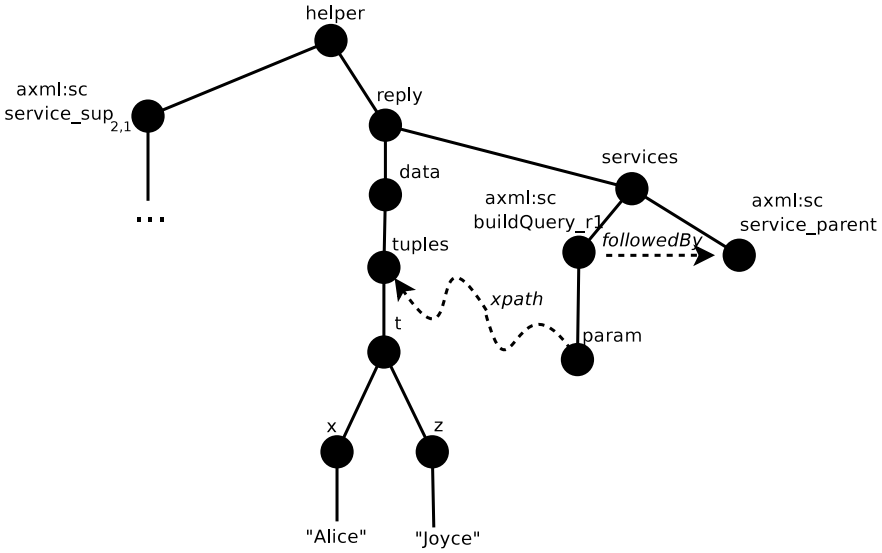


Figure 20: Helper document after calling the *buildQuery* service

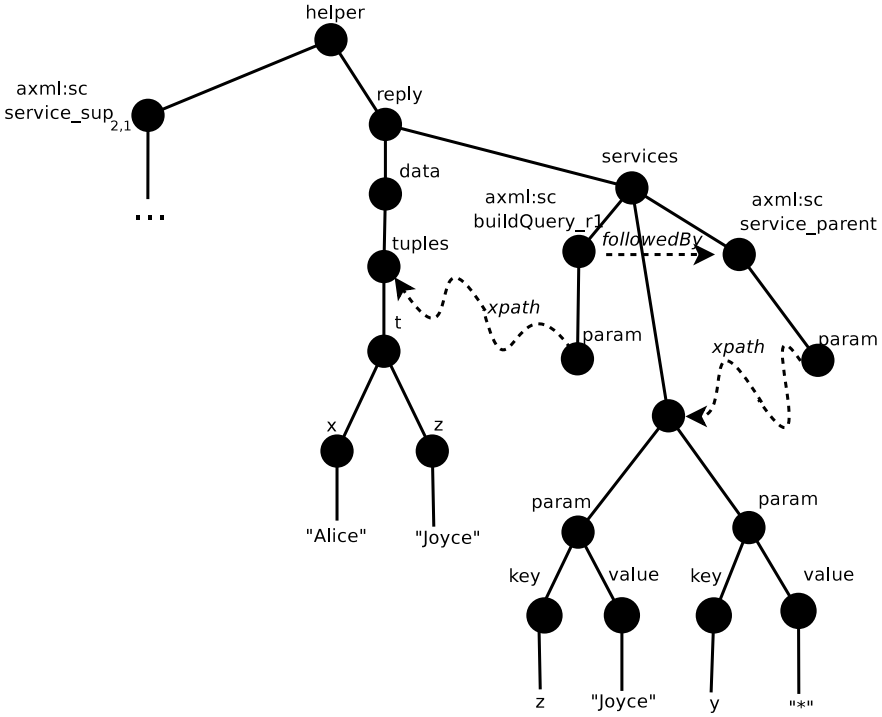


Figure 21: Helper document after calling *Service_parent*

- A call to *Rule_r1* is inserted to sup_{22}*.xml*, the AXML document that represents sup_{22}, using an X-OQL update service. For each relation we defined such a service, called *addRule*. When a rule is translated, in order to invoke *addRule*, the system creates a new AXML document, called *callAddRule*, that contains a service call to *addRule*. The service call is immediately materialized, invoking the *addRule* service.

Using Event-Sensitive continuous services In Section 3.2.3, when we discussed AXML services, we presented the mechanism of Event-Sensitive continuous services, that we developed and added to the AXML framework as an extension of continuous services. (The full details are given in Appendix A). The goal of this mechanism is to wake up a continuous service only when an event occurs, as opposed to the default behavior in which the service wakes up periodically. To make the system more efficient, we would like each service to wake up only when it potentially has new data for its clients. For instance, the service of relation R should wake up only after R receives new data. To announce that the event occurred, the application needs to call the *onServiceEvent* Web service, with the name of the service as a parameter. The *onServiceEvent* service is a part of our implementation of Event-Sensitive continuous services, and it is deployed with AXML.

We will continue with our rule

$r1 : sup_{22}@R(x, y)$:-$sup_{21}@P(x, z), parent@R(z, y)$

to demonstrate how dQSQ uses the mechanism.

- When the helper document receives new data from the last relation in the rule body, namely from $parent$, it should call *onServiceEvent*

for *Rule_r1*. As a result, *Rule_r1* will wake up and transfer the newly generated tuple/s to the relation in the head of the rule, sup_{22}. For this to happen, *Service_parent* returns, together with the data, a service call to *onServiceEvent* with the suitable parameters.

- Similarly, when sup_{22} receives new data, *Service_*sup_{22} should wake up, since there may now be new data for its own clients. Again, *Rule_r1* adds alongside the data sent to sup_{22}, a service call to *onServiceEvent*.

To conclude, Table 1 and Table 2 contain the AXML documents and services that are generated for each relation and rule, respectively, together with a short description of each one.

name	**description**
R.xml	The main AXML document of relation R, containing the tuples of the relation.
Service_R.xml	An X-OQL query service that returns data from R.xml according to the specified parameters.
ContService_R.xml	A continuous service that wraps *Service_R* as its base service.
Service_R_clients.xml	The clients database of *Service_R*. Contains the list of clients and the accumulated data sent to each one.
addRule_R.xml	An X-OQL update service that adds to *R.xml* a service call to deduce new facts by a rule. *addRule_R* is called once for each rule which relation R is in its head.

Table 1: The AXML documents and services generated for relation R

name	**description**
HelpRule_r1.xml	A helper AXML document that contains service calls to receive the relevant data from relations sup_{21} (first relation in rule body) and $parent$ (second relation in rule body).
buildQuery_r1.xml	A service that builds the parameter value for *Service_parent* based on the reply that was received from sup_{21}.
Rule_r1.xml	A service that pulls the relevant data from the helper document to sup_{22} (relation in rule head).
ContRule_r1.xml	A continuous service that wraps *Rule_r1* as its base service.
Rule_r1_clients.xml	The clients database of *Rule_r1*. (In this case there is only one client - sup_{22}.)
callAddRule_sup_{22}.xml	An AXML document that contains a service call to *addRule_sup_{22}*.

Table 2: The AXML documents and services generated for rule $r1$: $sup_{22}@R(x, y)$:-$sup_{21}@P(x, z), parent@R(z, y)$

3.4 Software Architecture

3.4.1 Programming Language

The programming language we chose for the implementation is *Java*. Since dQSQ is provided as an extension of the *Active XML* framework, and since *Active XML* is written in Java, Java was a natural choice for us. This is in addition to Java's other advantages such as being platform independent, its Object-Oriented approach and the provision of some useful libraries. The User Interface of the dQSQ system is also Java-based and implemented with *JSP* (Java Server Pages) [22], a powerful technology for building dynamic Web content.

3.4.2 System Structure

The system is composed of several components. Each component is a Java package (except for the UI, which is simply a collection of JSP files). Since the dQSQ system is distributed, each component operates on every peer. Below is a list of the components in the system, together with a short description, API and the input and output of each component.

1. **UI**: A collection of JSP files that form the Web user interface of the system. A JSP file is an HTML file with embedded Java code. When a JSP file is accessed from a Web browser, it is turned into a Java file, compiled (only in the first access) and loaded. We used this technology to build a dynamic Web user interface. The UI receives a request from the user, most commonly it is the query to be evaluated, and simply delegates the request to the *datalog2qsq* component by calling the suitable Web service.

2. **datalog2qsq**: A Java package that implements the dQSQ rewriting algorithm. Its input is composed of two parts:

   - When the system is initialized, this component receives the peer's part of the dDatalog program, which is simply kept in memory.
   - One of the peers receives the user query, which arrives at this component.

   The *datalog2qsq* component at the peer that started the query, performs the dQSQ rewriting process, turning the dDatalog program to a new set of rules. The component exposes Web services that receive the input from the JSP UI. For instance, the Web service *runQuery* is called to delegate the user query from the UI.
   The dQSQ rewriting is a distributed process. The *datalog2qsq* components at the peers that take part of the query evaluation send messages to each other via Web services. For example, when a peer creates a rule that should be located on a remote peer, it sends the rule to the remote peer by calling the *recvRuleFromRemotePeer* Web service.
   The output of this component is the set of rewritten rules. A rewritten rule, as well as the relations in it, are sent to the *qsq2axml* component to be translated to AXML. This is done by a direct invocation of Java methods.

3. **qsq2axml**: A Java package responsible for the translation of a rewritten dDatalog program to its representation in AXML. The API of this package is in a class called *Qsq2AxmlTranslator*, which exposes the methods *translateRule* and *translateRelation*. The *dat-*

alog2qsq component holds an instance of this class and calls these methods with the rules or relations to be translated to AXML.
The output of this component is naturally the AXML representation of the translated rules/relations, namely AXML documents and services. After creating an AXML document, the component adds it to the AXML documents repository by calling the AXML Web service *addDocument* at the current peer. Similarly, the Web service *addService* is invoked to add a new AXML service to the repository of services.

4. **axmlservices**: A Java package which contains Web services that are called directly by the AXML peer.
   Among the Web services in this package are:

   - *diff*: Pre-processes the reply of a continuous service before it is sent to the client. This service is automatically called by the AXML peer. (More details can be found in Section 3.3.2.)
   - *register*: Registers a new client of a continuous service in the database of clients. A call to this service is added by our translation mechanism before every call to a continuous service. (More details can be found in Section 3.3.2.)
   - *postRecvData*: Does all the required operations after a relation receives new data.

5. **termdetection**: Implements the termination detection algorithm (to be described in detail in section 4). As the termination detection is a distributed process, the *termdetection* component at all peers communicate with each other via Web services to exchange

messages.

Among the Web services in this package are:

- *isIdle*: Checks if the current peer is idle, and returns *true* or *false* accordingly.
- *recvAck*: Receives an acknowledgement confirming the arrival of a message.
- *incrementActivity* / *decrementActivity*: Increments/decrements the *activity* value of a relation. The *activity* variable stores the current activity level of each relation in the rewritten dDatalog program. It is a part of the mechanism that determines whether the query computation process is active or passive at any given moment.

6. **common**: Contains classes that are used by several components. For example, the classes *Relation* and *Rule* that are used by both *datalog2qsq* and *qsq2axml*. The former component creates a *Rule* object when generating a new rule during the rewriting, and the latter component translates the rule to AXML format.

7. **util**: Contains general utility classes, such as:

   - *Logger*: System logger, for error and debug messages.
   - *FileUtils*: Provides file-system services, such as copying and deleting files/directories.

Installing and Running dQSQ The system's components compile into a single jar file, *dqsq.jar*, that should be deployed to the AXML installation at every peer.

To make the dQSQ system more user-friendly, we provide two useful scripts for the user:

- *installdQSQ*: Installs the dQSQ system on top of AXML. The user should run this script only once. It deploys the dQSQ binaries on top of the AXML installation. This script also generates an HTML file, `dQSQHomePage.html`. Double-clicking this file takes the user to the dQSQ home page, where he can submit queries and perform other operations.
- *rundQSQ*: Runs the dQSQ system.

Summary Figure 22 illustrates the hierarchy of the major components, together with the data flow generated as a result of query submission. The phases of the data flow are as follows:

1. The user submits a query in the Web UI of the system at peer `P`.
2. The query is delegated to the *datalog2qsq* component, which starts the dQSQ rewriting process. The rewriting process may generate local and remote rules. Local rules are sent to the local *qsq2axml* component. Remote rules sent to the *datalog2qsq* component on the remote peer (peer `R` in this example) via a Web service, and the rewriting continues recursively.
3. The *qsq2axml* component translates the rewritten rule to AXML, generating a set of AXML documents and services. The new documents and services become active after adding them to the AXML repositories. This is achieved by calling the *addDocument*/*AddService* Web services.

4. The AXML documents and services are activated in the AXML peer. The AXML peer calls some Web services in the *axmlservices* component, such as the *diff* service, which pre-processes the reply of a continuous services, namely it gets as input a reply and returns the processed reply.

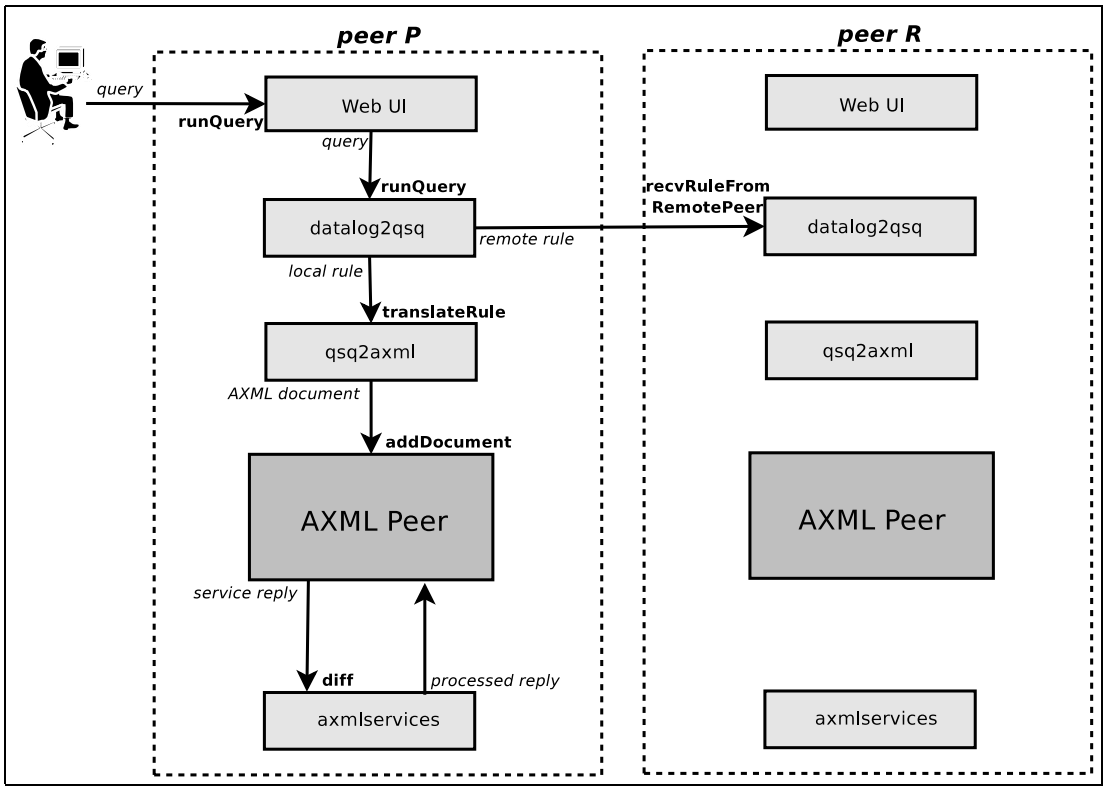


Figure 22: dQSQ system architecture and data flow

4 Termination Detection

In this section we describe the termination detection mechanism that we defined and implemented for the dQSQ system. The objective of the mechanism is to detect when the query evaluation process is finished. We begin in Section 4.1 with a theoretical background that includes a description of a general distributed network, and the algorithm we defined for such a network. We also explain how we match a dQSQ program to the model. Such a match yields a valid termination detection algorithm for our dQSQ system. Next, in Section 4.2 we describe the implementation of the algorithm in our AXML environment, which extends the AXML representation of dQSQ programs that was shown in Section 3.3.

4.1 Theoretical Background

4.1.1 The Model

We define here a general model of a distributed network on which some computation process is running. Later on we will show how we applied this model on the dQSQ system. Our network consists of `n` nodes. Each node can be connected to one or more nodes via a directed edge. If there is an edge from r_1 to r_2, we say that r_2 is a *neighbor* of r_1. We assume that one node, that we call the *leader* initiates the computation by sending a message to one or more of its neighbors. When a node receives a message, it processes the message and sends one or more messages to its neighbor nodes. A node has two possible states: *active* and *passive*. After node r_2 receives a message from r_1, r_2 becomes active and sends back an acknowledgement to r_1. r_2 remains active until (1)

it finished processing all the messages it received and (2) it received an acknowledgement for all the messages it sent to other nodes.

Definition 4.1 *We say that the computation process is active if one of the following holds:*

1. *There is a node that has some "duties" to perform, e.g., sending an acknowledgement on a received message, or some processing to do, due to such a received message.*
2. *There is a node that processes a message.*
3. *There is a message on its way from one node to another.*

Lemma 4.2 *As long as the computation process is active, there is at least one active node.*

Proof. According to Definition 4.1, as long as the computation process is active, at least one of the following holds:

1. There is a node r_2 that has some "duties" to perform. This is the case in which there is a message that arrived from another node, r_1, and waits for r_2 to read it. According to the model, after r_2 receives a message, it first changes its status to *active*, then it sends back an acknowledgement to r_1, and then it starts processing the message. There are two options: (a) r_2 changed its status to *active*, and did not start processing the message yet. In this case r_2 is obviously active. (b) r_2 did not change its status to *active* yet. In this case r_2 did not send an acknowledgement to r_1 yet, and therefore r_1 is active[3].

[3]Recall that according to the model, a node remains active until it receives an acknowledgement for all the messages it sent to other nodes

2. There is a node that processes a message, in which case this node is currently active.

3. There is a message on its way from one node to another. If a message is currently on its way from node r_1 to node r_2, then node r_1 is currently active and will remain active until it receives an acknowledgement from node r_2.

□

From Lemma 4.2 we conclude directly:

Lemma 4.3 *If at time* `t` *all nodes are passive, the computation process is finished at time* `t`.

4.1.2 The Algorithm

The termination detection algorithm that we used is partially based on the algorithm presented in [16]. It is summarized in pseudo-code in Figure 23 and described below.

The node that initiates the computation process is designated as the leader. The procedure *isFinished* is called periodically. *isFinished* checks if the leader node is idle by calling *isIdle* with the current cycle ID, which is stored in the variable `cycle_id`, and the leader node as parameters (line 5). We say that the node is *idle* if (1) the node is currently not active (line 8) and (2) all its neighbors (if such exist) answer that they are idle (line 14).

The current cycle ID is incremented whenever *isFinished* is called (line 4). The cycle ID is maintained in order to avoid endless recursions in calls to the procedure *isIdle* as a result of loops in the graph. Note that

```
 1.    integer: idleness:= 0;
 2.    integer: cycle_id:= 0;

 3.    procedure boolean  isFinished
 4.       cycle_id++;
 5.       return isIdle(leader,cycle_id);
 6.    end proc;

 7.    procedure boolean  isIdle(node, cycle_id)
 8.       if (node is active)
 9.          return false
10.       end if;

11.       if (node already participated in this cycle)
12.          return true
13.       end if;

14.       for each neighbor of node
15.          if (isIdle(neighbor, cycle_id) = false)
16.            return false
17.          end if;
18.       end for;

19.       // if we reached this code, then all of the
20.       // node's neighbors (if such exist) are idle
21.       idleness++;
22.       if (idleness > 1)
23.          return true
24.       else
25.          return false
26.       end if;
27.    end proc;

28.    procedure  onReceiveMessage
29.       idleness = 0;
       ...
30.    end proc;
```

Figure 23: The termination detection algorithm

if the node has already participated in this cycle it answers positively (line 12), so that only the first participation in the cycle is effective.

In order to "remember" how long a node has been idle (according to the definition of *idle* above), it keeps the variable `idleness`. The procedure *isIdle* increments `idleness` if the node is idle (line 21). Only if the node is idle for two cycles in a row (the `idleness` value of the node is greater than 1), the node answers positively (line 23). Whenever a node receives work, meaning it receives a message during the computation process, it resets the `idleness` value to 0 (the procedure *onReceiveMessage* - line 29). Intuitively, we need all nodes to be idle for two cycles in a row in order to detect termination, because we need them to be idle in the *period* between two consecutive cycles. The formal explanation is given in the proof of Theorem 4.4.

If the leader receives a positive answer from all its neighbors and has itself been idle since the previous cycle, then it concludes that the computation is finished.

Theorem 4.4 *The procedure isFinished returns true if and only if the computation process is terminated.*

Proof.

1. *If the computation process is terminated, isFinished returns true:*
   If the computation is terminated at time `t`, then all nodes are passive at time `t`. Thus all nodes will answer positively and as a result *isFinished* will return true at time `t+c`, where `c` is some constant.

2. *If isFinished returns true, the computation process is terminated:*
   Suppose cycle `k` started at time t_0, and cycle `k+1`, to which all nodes responded positively (*isIdle*=true), started at time t_1. Let `X`

be one of the nodes in the network. Suppose X received the *isIdle* message of cycle k at time t_{0x}, and the *isIdle* message of cycle k+1 at time t_{1x}. We know that X responds true at t_{1x}, which means its idleness value is at least 2. From this we conclude that:

(a) (†) The idleness value of X became 1 at t_{0x} (because the idleness value is increased only when receiving an *isIdle* message).

(b) X did not receive any computation message (a message as a part of the computation process) in [t_{0x}, t_{1x}], otherwise its idleness value would not become 2 at t_{1x}. This is true since idleness is set to 0 in the procedure *onReceiveMessage*.

We claim that X is passive at time t_1. Since X did not receive any computation message in [t_{0x}, t_{1x}], X can be active at t_1 only if it received a message before t_{0x} and it still processes this message at time t_1. But if this were true, the idleness value of X would not be increased at t_{0x} (because idleness value is not increased if the node is active) - in contradiction to (†). From this we conclude that X is passive at t_1. Since for each node j, there are such t_{0j} and t_{1j} ($t_{0j} < t_1 < t_{1j}$), we conclude that all nodes are passive at time t_1. Now, according to Lemma 4.3, the computation is finished at time t_1.

□

4.1.3 Applying the Algorithm on dQSQ

We now describe how we applied the termination detection model and algorithm on the dQSQ system. Recall that dQSQ rewrites the original

dDatalog program to a new program according to the given query. We applied the termination detection model on the rewritten program such that:

1. Every relation in the rewritten program is a node in the network.
2. For each rule in the rewritten program, there is an edge from the relation in the head of a rule to every relation in the body of that rule.

As an example, the directed graph in Figure 24 describes the network for the rewritten dDatalog program in Figure 5.

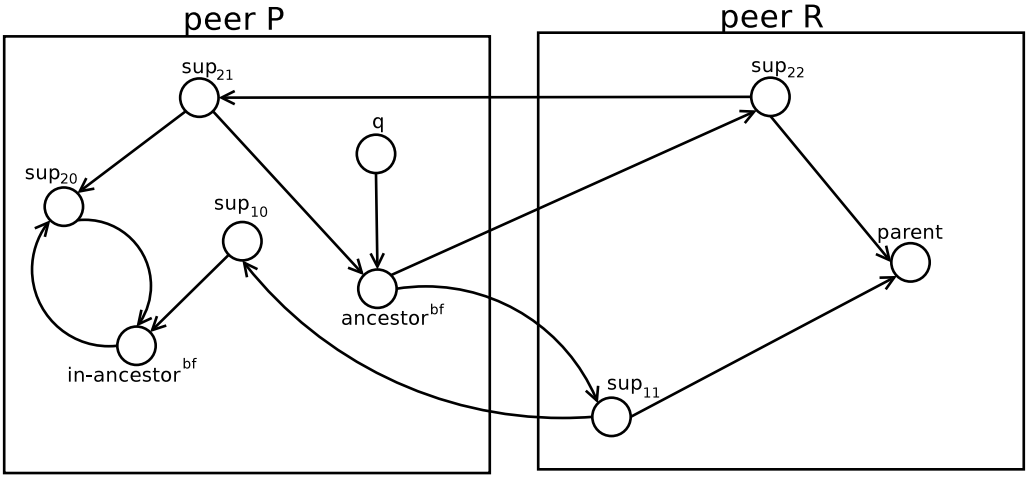


Figure 24: A network corresponding to a rewritten dDatalog program

The relation in the head of the query is selected to be the leader node. The graph in Figure 24 was generated as a result of submitting the query $q@P(x)$:- $ancestor@P(\text{"}Joyce\text{"}, x)$, thus the leader is node $ancestor^{bf}$. The leader wakes up periodically[4] and executes the *isFinished* procedure from the algorithm in Figure 23.

[4]The period is configurable using the *frequency* attribute of AXML service calls. In the implementation we defined the default value to be 10 seconds.

As will be described next in Section 4.2, the termination detection mechanism that we implemented for our dQSQ system, has the properties of the general model that was defined in Section 4.1.1. Therefore, the termination detection algorithm presented above is valid for our dQSQ system.

4.2 Termination Detection Using AXML

Section 4.1 described the termination detection algorithm, the model and the way we applied the model on the dQSQ system. We now focus on the implementation details.

Building the network graph As mentioned in Section 4.1.3, our termination detection graph correlates to the rewritten dQSQ program, such that every relation in the rewritten program is a node in the graph, and for each rule in the rewritten program, there is an edge between the relation in the head of the rule to every relation in the body of that rule. Since the relations and rules are distributed among several peers, every peer holds only a part of the graph, as illustrated in Figure 24. In our implementation, each peer holds a local map of its part in the graph. The map is actually a mapping between every node to a list of its neighbor nodes. An important observation is that the map is built dynamically, while the dQSQ rewriting process is running. When a rule of the form *r:-p,q* is generated, the relations `r`, `p` and `q` are added as nodes in the graph, unless they already exist in it, and two edges are added: one edge from `r` to `p` and another edge from `r` to `q`. Table 3 gives the map of peer `P` of the *Ancestor* program in Figure 5 in its final state.

relation	neighbors
q	$ancestor^{bf}@P$
$ancestor^{bf}$	$sup_{11}@R$, $sup_{22}@R$
$in\text{-}ancestor^{bf}$	$sup_{20}@P$
sup_{10}	$in\text{-}ancestor^{bf}@P$
sup_{20}	$in\text{-}ancestor^{bf}@P$
sup_{21}	$sup_{20}@P$, $ancestor^{bf}@P$

Table 3: Peer P's map of the network graph

Selecting the leader Recall that the relation in the body of the query is selected to be the leader node. The leader wakes up periodically and checks if the query computation is finished, by running the algorithm in Figure 23. The dQSQ rewriting process identifies the leader node and inserts a periodical service call to the Web service *isFinished* in the AXML document of the corresponding relation. Figure 25 extends the structure of the relation $ancestor^{bf}$ in Figure 11, after inserting the *isFinished* call. The query in our case is $q@P(x)$:- $ancestor@P(\text{"}Joyce\text{"}, x)$, thus the leader node is indeed $ancestor^{bf}$.

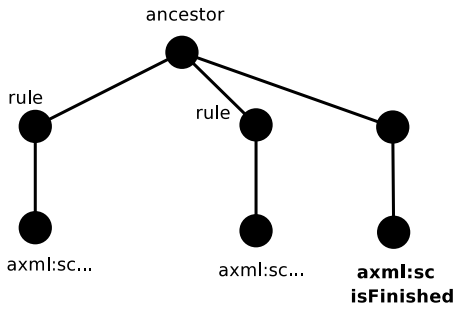


Figure 25: Inserting the *isFinished* service call

Determining node activity The termination detection algorithm in Figure 23 is based on the ability of each node in the graph to answer whether it

is active or passive at the current moment. For this purpose the system defines an `activity` variable of type integer for every node. The value of `activity` can be 0 or greater. `activity` of 0 indicates that the node is currently passive, while a value greater than 0 indicates that the node is currently active. If the node is active, then its `activity` value can be referred as its activity level.[5] We will demonstrate the change in the `activity` value with the example illustrated in Figure 26.
We assume that relation `r` has two registered clients - `c1` and `c2`. When relation `r` receives new data from relation `p` (step 1), it increments the `activity` value by 3 - once for itself and one more time for each of its clients. `r` does some processing on the data and then sends an acknowledgement to `p`, in addition to decrementing the `activity` value by 1 (step 2). Afterwards, `r` checks if it has new data to send to its clients. For client `c1` it has no data, thus the `activity` value is decremented by 1 immediately (step 3). For client `c2`, however, `r` has new data to send, so the `activity` value remains 1 until `c2` sends an acknowledgement (step 4). Observe that the state of `r` is active from step 1 through step 4, since its activity value is greater than 0. Therefore, if `r` is asked whether it is idle at that time, it will answer negatively.

While from an algorithmic viewpoint the above process is rather simple, its implementation in AXML was a challenge, because AXML does not provide an easy way to add user-defined logic to the usual functionality of materialization of embedded service calls. We overcame this problem by doing the following:

1. Implementing the desired functionality in Java.

[5] Intuitively, as the activity value gets higher, the node is more active.

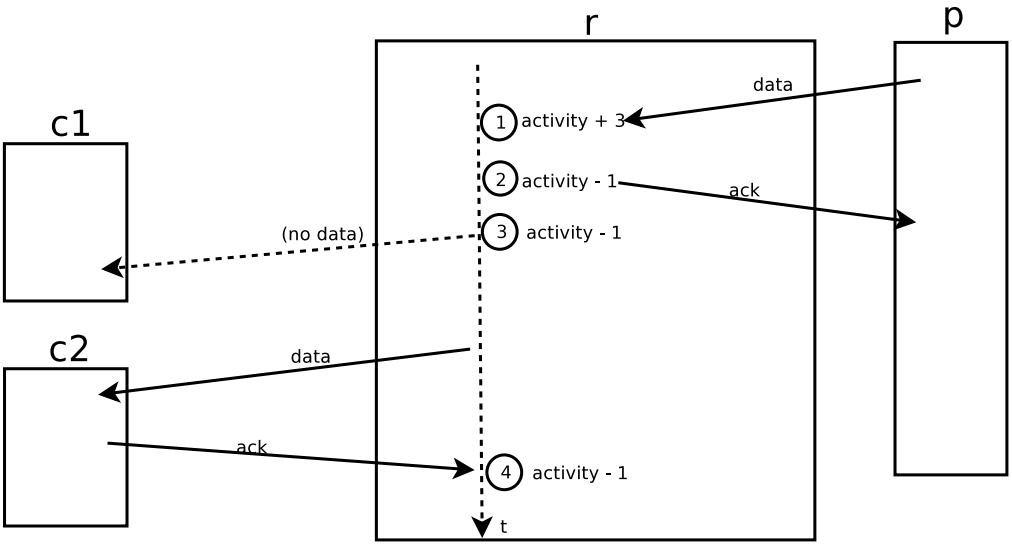


Figure 26: Example of determining node activity

2. Wrapping the Java code as a Web service.

3. Inserting to AXML documents service calls to the Web service, whenever we need to perform the desired operation.

For example, when a relation receives new data, the service returns, apart from the data itself, a service call to the special Web service *postRecvData*. This service wraps a Java method, which does all the required operations after receiving the data, such as incrementing the `activity` value. This is illustrated in Figure 27 that extends Figure 12 with the *postRecvData* service call.

Managing acknowledgements In the correctness of our termination detection algorithm, we assumed that every message is acknowledged (see Lemma 4.2). Here is a description of the implementation of this acknowledgments mechanism in our system.

- We implemented a Web service called *recvAck* and deployed it on

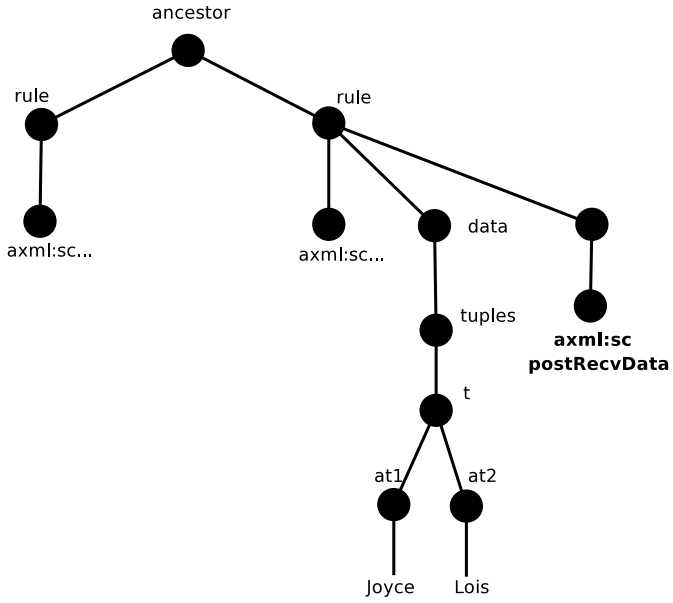


Figure 27: Inserting the *postRecvData* service call

every peer. The service receives the name of the node representing the relation that should receive the acknowledgment as a parameter. *recvAck* does all the required operations after receiving the acknowledgment, such as decrementing the activity value of the receiving relation.

- After the node of a relation receives a message, it should send an acknowledgment to the sender of the data. This is achieved by inserting a service call to *recvAck* in the suitable AXML document, next to the data that was received. For example, recall the AXML helper document in Figure 20, which corresponds to the rule $sup_{22}@R(x, y)$:-$sup_{21}@P(x, z), parent@R(z, y)$. Figure 28 extends the helper document with the additional call to the *recvAck* service. After the helper document, that represents relation sup_{22} at peer R receives the data from relation sup_{21}, it calls the *recvAck* service at

peer P with sup_{21} as a parameter. (The order of the calls is dictated by the *followedBy* relationship between the service call nodes.)

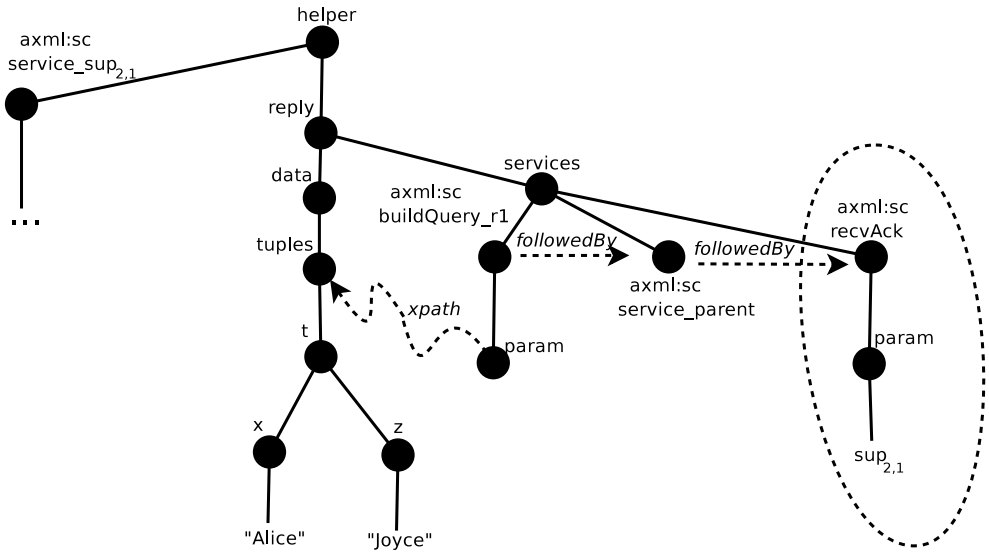


Figure 28: Adding the *recvAck* service call to the helper document

5 A Guided Tour

The system provides a friendly Web-based User Interface, implemented in JSP (Java Server Pages) [22]. In this section we give a visual tour of the system. We will take as an example our *Ancestor* dDatalog program in Figure 3.

Before running the dQSQ system, the user should define the dDatalog input program. Each peer should receive its own part of the dDatalog program as an XML file. This file, which has a reasonably straightforward format, contains a definition of the extensional relations as well as the rules that are located on the peer. Next, the dQSQ system is initialized at all peers. For this, the user should simply run a script which is provided with the system. The full details were described in Section 3.4.

After the system has been initialized, the user may access the Web UI of the system. Figure 29 shows the main screen of the system at peer `P`.

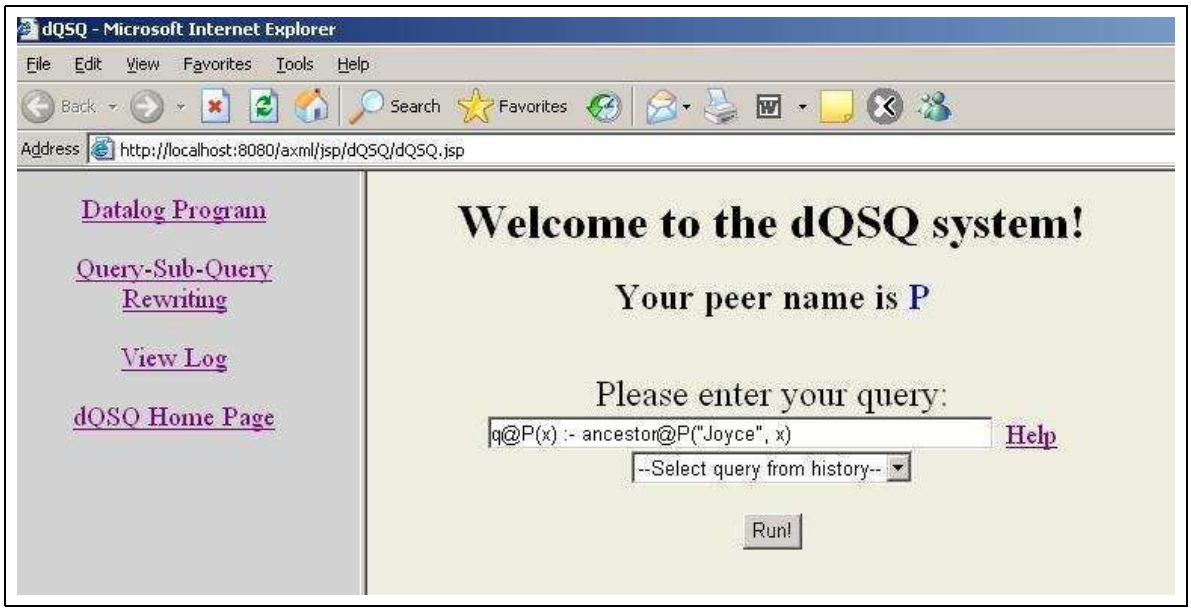


Figure 29: dQSQ: Main screen

You can see that the main screen is divided to two parts:

1. The left frame is a menu that enables the user to perform the following operations:

   - View the input Datalog program on this peer i.e., this peer's part of the distributed Datalog program.
   - View the QSQ rewriting of the program on this peer. Clicking this link will show a list of the rules that were generated by the dQSQ rewriting process for the query. The list of rules is built incrementally during the rewriting, and at any given moment the user can view the rules that have been generated so far. Figure 30 shows the full dQSQ rewriting of peer `P` for the query `q@P(x):-ancestor@P("Joyce",x).`
   - View the peer's log.
   - Go to the dQSQ home page.

2. The main frame is where the user submits queries. The user can write his own query, or select and edit a previous query (if such exists) from a list. A click on the 'Help' link will open a window with guidelines for writing queries, including explanations about the syntax. The query that was submitted in Figure 29 is `q@P(x):-ancestor@P("Joyce",x).` A click on the 'Run!' button will submit the query. Consequently, the systems at all relevant peers operate, perform the dQSQ rewriting process and start evaluating the query. The main screen then changes to a new screen that shows the results of the query, as shown in Figure 31. The results are shown incrementally, namely at any given moment the

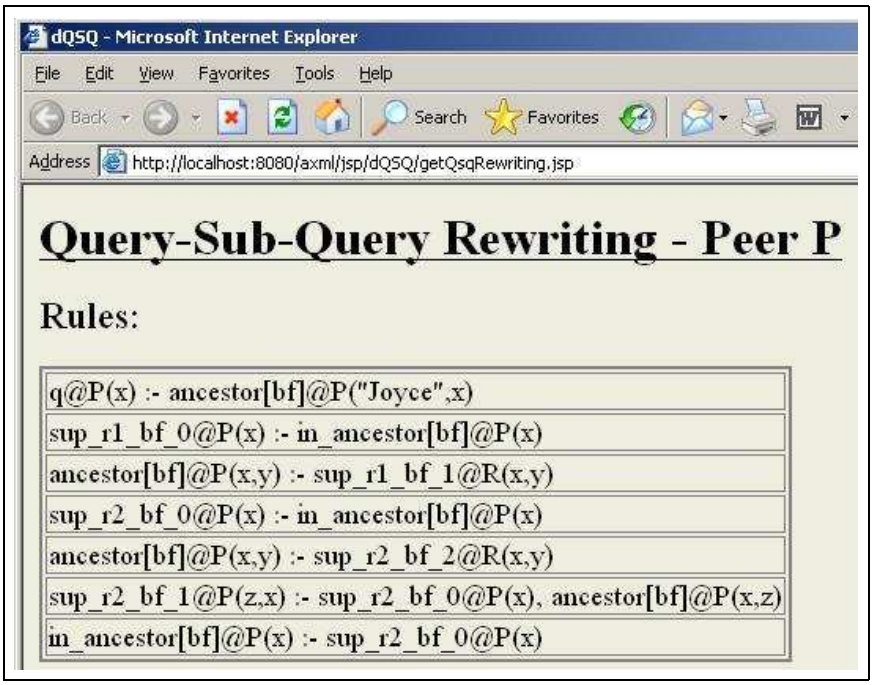


Figure 30: dQSQ: Viewing the dQSQ rewriting

user can view the results that have been obtained so far.

The results screen is divided to two parts:

- The upper part shows the accumulated results of the query. A click on the 'Refresh Results' button will refresh the results, adding the new results obtained by the continuous query computation process. Pressing the 'Stop Query' button will stop the query while it is being evaluated, taking the user back to the main screen.
- The lower part of the screen is a board with notifications about the query computation status i.e., whether it is still active or if the computation is finished. If the query computation is finished, the results on the screen are the final query results. The user can then submit a new query by clicking the 'Run new query' link. This is shown in Figure 32.

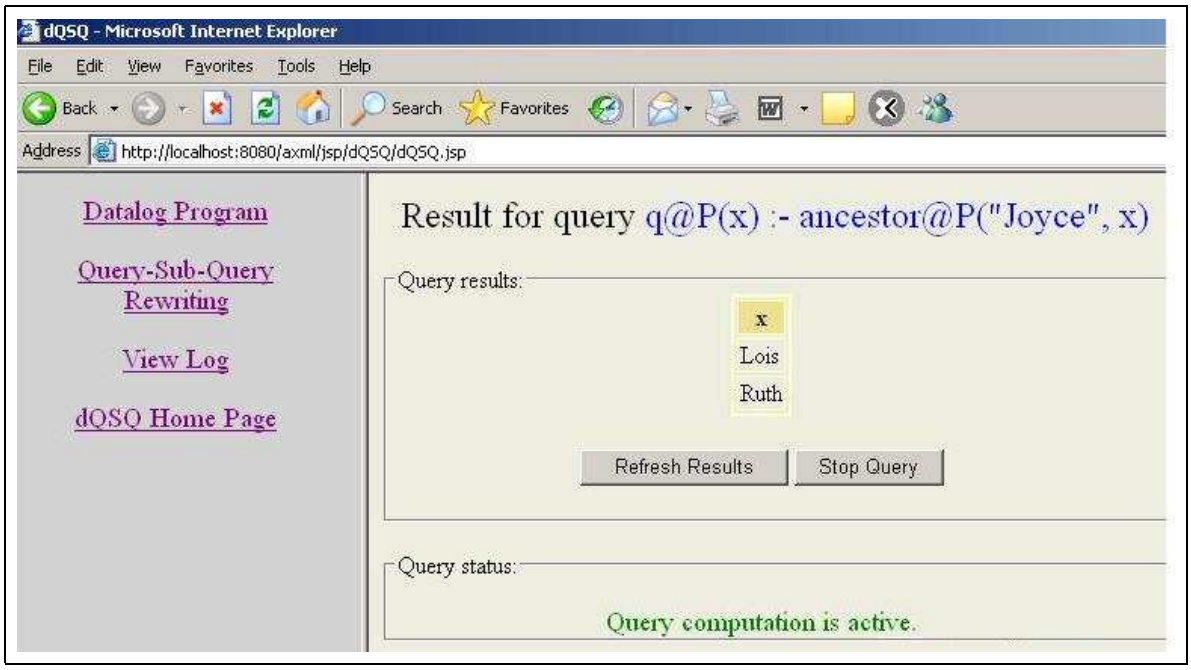


Figure 31: dQSQ: Query result screen

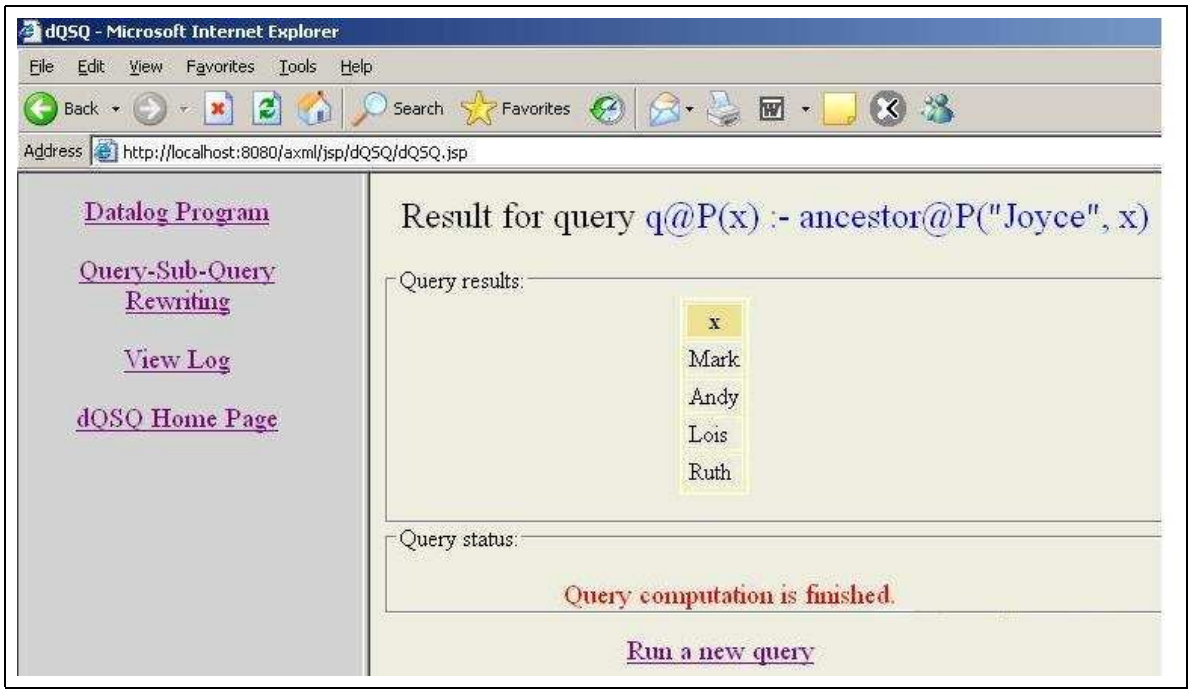


Figure 32: dQSQ: Query's final result

6 Related Work

In this work we have built a system that optimizes Datalog query evaluation in a Peer-To-Peer (P2P) environment. The problem of distributed query processing in a Web context is very active [14] and falls under the field of distributed data management [28] in general.

We based our work on the underlying theory presented in [2], which shows how to adapt *Query-Sub-Query (QSQ)* [6, 32], an old technique from deductive databases, to a setting where autonomous and distributed peers share large volumes of interrelated data.

Our overview of the related work begins in Section 6.1 with the technology that we used in the implementation of the system. We then consider Datalog and query optimization techniques in Section 6.2. Finally, in Section 6.3 we consider related work in the field of termination detection in a distributed setting, which was an important aspect of our work.

6.1 XML, Web Services & P2P

Our system is implemented using *Active XML (AXML)* [7, 3], a system based on exchange of *XML* documents with embedded calls to *Web services* in a *P2P* environment. AXML documents are XML documents where some of the data is given explicitly while other parts are given only intensionally by means of embedded calls to Web services, that can be called to generate the required information. An overview of AXML was given in Section 3.2. We will now overview the technology which forms the basis of AXML.

XML and Web services XML [34] is a data model and representation format for semi-structured data [5], which raised a considerable interest among the data management community as a standard for data exchange between remote applications, notably over the Web. Unlike HTML, XML can meaningfully represent tree-structured data, by allowing arbitrary node labels (i.e., element "tags"). Most importantly for data management, XML comes equipped with high level query languages, e.g., XQuery [37] and XPath [36], and flexible schema languages, e.g., XML Schema [35]. The publication of XML data and the access to it are facilitated by Web services, which are network-accessible programs taking XML parameters and returning XML results. The WSDL [33] and SOAP [31] standards enable describing and calling these remote programs seamlessly over the Internet. In the AXML approach, XML and Web services are effectively leveraged for complex data management tasks.

P2P architecture Centralized architectures for data integration somehow contradict the essence of the Web, which is based on the autonomous nature of Web systems. Furthermore, centralized architectures hardly scale up to the large size of the Web. P2P architectures propose an alternative to centralized ones, and are already spreading, notably in the context of file-sharing (e.g., see [23, 17, 13]). P2P architectures capture the autonomous nature of the systems participating in them, and the ability of these systems to act both as producers of information (i.e., as servers) and as consumers of information produced by others (i.e., as clients). In the AXML approach, a P2P architecture forms the basis for scalable distributed data management. P2P information management [1], is becom-

ing popular with file-sharing systems and a number of techniques have been developed to support it, e.g., distributed look-up as in [10, 19].

6.2 Datalog & Query Optimization

Optimization of evaluation of Datalog queries in a centralized setting is a field that has been thoroughly studied and researched. The various techniques for recursive query processing are usually separated into two classes, depending whether they focus on *top-down* or *bottom-up* evaluation [6, 20].

Top-down & bottom-up strategies The top-down approach is goal-directed. It starts with the goal (i.e., the query to evaluate), which is reduced to subgoals, or a number of simpler problems, until a trivial problem is reached. Then, the solution of larger problems is composed of the solutions of simpler problems until the solution of the original problem is obtained [20]. The bottom-up computation, on the other hand, starts with known facts and extends the set of known "trues" using rules. Thus, it can derive new facts from old facts and rules. This extension continues until the solved problem is present in the computed set of facts [12].

QSQ and *Magic sets* are two similar techniques that combine the top-down and bottom-up approaches. QSQ and its extension to a distributed setting were described in Sections 2.1.3 and 2.2.3, respectively. The essence of QSQ is a rewriting of the Datalog program according to the given query, based on the propagation of the bindings of variables. It is stated in [2] that QSQ computes the correct answer to the query and materializes only a *minimal* set of tuples. The QSQ technique is elaborated in [6, 32].

Magic sets [11, 6], like QSQ, is based on rewriting of the Datalog program so that bottom-up fixpoint evaluation of the program avoids generation of irrelevant facts. The major advantage that QSQ and Magic sets provide is that they allow a bottom-up computation to be focused with respect to the query, thus improving the efficiency of answering queries [24].

Optimizations in distributed settings QSQ-like optimizations for query evaluations over *distributed* deductive databases are considered in [16, 21]. Both works present similar techniques for optimization. The problem is formulated as a system of cooperating processes communicating by message passing. A graph that expresses the decomposition of the original query into sub-queries is constructed, and the "sideway information passing" strategy is used to restrict the computation to produce relevant, or at least potentially relevant facts. These papers are from the pre-Web era, and the architecture considered in them is not P2P. In addition, they are theoretical and, to the best of our knowledge, without any implementation. One of the major contributions of the present work is to adapt the query processing to a P2P setting in the Web, and to find a suitable implementation in this environment.

The idea of *pushing queries* to data sources is a fairly standard technique in mediator systems [29], and involves issues such as verifying that the remote source is capable of evaluating them [30].
In [25], the problem of logical data integration is studied. The paper presents a methodology for integrating relational and tree-structured data sources, in particular XML documents, under a single XML global schema. In this approach a user query is formulated in terms of the

global schema; the system translates the query into sub-queries expressed in terms of the local schema; and the sub-queries are pushed to the local data sources.

[18] surveys query rewriting using views, a problem that arises in data integration systems. Users of such systems pose queries in terms of mediated schema. The mediator system reformulates the query into a query that refers directly to the data source schemas.

The pushing of queries has also been studied in the context of AXML. For instance, it was shown in [4] how to push a sub-query to a function call, when the result of the call may contain more data than is actually needed for evaluating the query. Such a technique reduces data transfer and computation.

The technique proposed in the current research can be applied in the context of pushing queries, and expected to be particularly useful in cases where there are *recursive* integration definitions. It will be interesting to research optional usages of dQSQ for this kind of applications.

[26] considers a different aspect of optimization. This work presents a *semi-join* technique to minimize communications. The semi-join technique from conventional databases is adapted to distributed databases that support logic programming, in particular recursive rules.

6.3 Termination Detection

Termination detection is a very prevalent problem in distributed computing. We borrowed the main ideas from the termination detection algorithm in [16], a work that also deals with optimization of query evaluations over distributed deductive databases. As previously mentioned, [16] formulates the query optimization problem as a distributed com-

putation in a network of processes communicating by messages. The network is built dynamically, and its structure depends only on the intensional database, namely the set of rules. The termination detection algorithm is designed to operate on the generated network. This mechanism resembles the one we used in our work (see Section 4.1.3), in the sense that our termination detection algorithm also runs on a network which is built dynamically according to the intensional database and the given query. There is a difference, however, in the structure of networks. Our network reflects the rewritten dDatalog program as a result of the dQSQ optimization, while the network in [16] naturally suits the optimization technique presented there.

7 Conclusions and Future Work

We have presented *dQSQ*, a distributed algorithm which optimizes evaluation of queries over distributed Datalog (dDatalog) programs. dQSQ is based on the extension of the Query-Sub-Query (QSQ) technique for evaluation of centralized Datalog queries for deductive databases. We defined a mechanism to represent dDatalog programs using Active XML. We also defined a termination detection mechanism and showed how to apply it when building a system based on the AXML representation. Finally, we developed a dQSQ system, based on these mechanisms. The system implements a distributed QSQ rewriting of the dDatalog program based on the given query, translates the rewritten program to AXML and includes a termination detection process that checks if the query computation is finished. Our system also includes a dynamic Web-based user interface, that provides the user an easy way to interact with the system.

It will be interesting to use the dQSQ system for real applications. For instance, it will be a challenge to build a system, which uses dQSQ for the telecommunication diagnosis problem that was presented in [2]. Such a system will transform the input Petri net, which describes the telecom network, to a dDatalog program i.e., to a set of relations and rules as described in [2], and then provide the dDatalog program as the input to the dQSQ system. Note, however, that if implemented precisely as defined in [2], this will require two additions to our dQSQ implementation: allowing the presence of (1) function symbols and (2) negation in dDatalog programs. Such a support was not in the scope of our work.

An interesting subject for future research is to find other problems that can be stated in terms of query evaluation in deductive databases. Once a problem is modeled by Datalog, it can benefit from the optimization provided by the dQSQ system. A first step in this direction was already given in [2], which showed that in addition to the specific telecommunication diagnosis problem that was presented there, dQSQ allows solving a much larger class of system analysis problems. The paper also mentions the potential of the deductive database paradigm in the context of Peer-To-Peer networks, which motivate recursive data management.

One of the challenges of this work was to explore whether it is feasible to use AXML as the platform for our system. We successfully managed to build such a system, but along the way we had to deal with many obstacles. On the one hand, using AXML was an advantage because it spared us the need to deal with the communication layer and infrastructure. Furthermore, AXML provides some useful built-in mechanisms such as *continuous services*, which enable the server to asynchronously send a stream of messages as an answer . On the other hand, it raised several difficulties, such as missing functionalities and performance problems. It will be interesting to implement a dQSQ system that is not based on AXML, and to compare it to our system in terms of performance, ease of implementation and ease of use. One optional direction is to build such a system from scratch, using pure Web services for communication between the peers. With this approach, however, we will lose all the advantages gained when using AXML, as mentioned above.

During the implementation, we contributed significantly to the AXML open-source code. Our contribution includes bug fixes, performance im-

provements, system stabilization and implementation of new features. All software is available in the AXML open source [9] within the ObjectWeb framework [27].

There are still, however, some open problems in AXML that we did not resolve in the scope of this work.

Among these problems are the following:

- A severe performance problem when the system is in use for several queries in a row.
- Lack of transaction management and concurrency control mechanisms. If these are added to AXML, it will solve problems caused by simultaneous updates of documents.

These problems harm the performance and stabilization of the dQSQ system, and are currently considered as limitations. We have implemented some workarounds to reduce the effect of these problems on our system, yet we did not manage to eliminate them completely. Further research in this direction in attempt to solve these issues will benefit our system, as it will make it more robust and stable, and might improve the system performance, especially when the rewritten program is large.

Another interesting research direction is to measure the improvement obtained from dQSQ compared to naive distributed query evaluation. In addition, it will be interesting to check how other optimization techniques, besides QSQ, can be employed on Datalog programs in a distributed setting, and to compare such techniques to dQSQ.

Finally, the dQSQ system supports the submission of several queries sequentially, but it does not support evaluation of simultaneous queries. Allowing multiple queries to run together (from a single peer or from

different peers) will clearly be a useful feature. We believe that such a support is attainable with only little changes in the current system. The main challenge here is to handle the generated load on the system as a result from the multiple queries.

References

[1] S. Abiteboul. Managing an xml warehouse in a p2p context. In *15th International Conference on Advanced Information Systems Engineering*, 2003.

[2] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *Proc. of PODS*, 2005.

[3] S. Abiteboul, O. Banjelloun, and T. Milo. The Active XML project: an overview, technical report, Gemo, 2004.

[4] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active xml. In *Proc. of SIGMOD*, 2004.

[5] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

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

[7] The Active XML homepage.
`http://activexml.net`.

[8] V. Aguilera. The X-OQL homepage.
`http://www-rocq.inria.fr/~aguilera/xoql`.

[9] Open Source Active XML.
`http://forge.objectweb.org/projects/activexml`.

[10] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *CACM*, 46(2):43–48, 2003.

[11] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to execute logic programs. In *Proc. of PODS*, 1986.

[12] R. Bartak. Guide to prolog programming, 1998.
`http://ktiml.mff.cuni.cz/~bartak/prolog/intro.html`.

[13] The BitTorrent Homepage.
http://www.bittorrent.com.

[14] R. Braumandl, M. Keidl, A. Kemper, and D. Kossmann. Objectglobe: Ubiquitous query processing on the internet. *VLDB Jour.*, 10:48, 2001.

[15] G. Cobena, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proc. of ICDE*, 2002.

[16] A. Van Gelder. A message passing framework for logical query evaluation. In *Proc. of SIGMOD*, 1986.

[17] The Gnutella homepage.
http://www.gnutella.com.

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

[19] M. Harren, J. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. *Peer-To-Peer Systems Int. Workshop*, 2002.

[20] Y. Hinz. Bottom-up is the trend, 2002.
http://faculty.ed.umuc.edu/~meinkej/inss690/hinz.pdf.

[21] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proc. of VLDB*, 1989.

[22] JSP - Java Server Pages technology.
http://java.sun.com/products/jsp.

[23] The Kazaa Homepage.
http://www.kazaa.com.

[24] D. Kemp and P. Stuckey. Magic sets and bottom-up evaluation of well-founded models., 1991. (Paper) Bull Information Systems, Phoenix, AZ.

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

[26] W. Nejdl, S. Ceri, and G. Wiederhold. Evaluating recursive queries in distributed databases. *TKDE*, 5(1):104–121, 1993.

[27] ObjectWeb, Open Source Middleware. `http://forge.objectweb.org`.

[28] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

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

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

[31] Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP`.

[32] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. *In Proc. Int. Conf. Expert Database Syst., L. Kerschberg, Ed., Charleston*, 1986.

[33] Web Services Definition Language (WSDL). `http://www.w3.org/TR/wsdl`.

[34] The Extensible Markup Language (XML) 1.0 (2nd Edition). `http://www.w3.org/TR/REC-xml`.

[35] The XML Schema specification. `http://www.w3.org/TR/XML/Schema`.

[36] XPath 1.0: An XML Path Language. `http://www.w3.org/TR/xpath`.

[37] XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`.

A Changes & Fixes in Active XML

We describe here our contribution to the AXML open-source code.
During the implementation of the dQSQ system, we experienced some difficulties with AXML.
The problems we encountered in AXML can be divided to three categories:

1. Bugs.
2. Missing functionalities.
3. Performance problems. These were particularly severe in the context of continuous services. The fact that a continuous service wakes up periodically and invokes the service for each client, generated a heavy load on the system, especially when one service has several clients registered on it. The *Event-Sensitive Continuous Services* mechanism that we implemented (details will follow) improved the performance significantly.

Below is a list of our main additions/fixes to AXML:

- **Event-Sensitive continuous services**. Prior to our work, the behavior of a continuous service was to wake up periodically, reevaluate the client's request and return an updated result. This behavior may suffice when the result of the service is not expected to change frequently. In the dQSQ system, however, service results may change instantly, as the query computation is a continuous process. With Event-Sensitive continuous services, a client who registers to a continuous service may ask from the server to reevaluate the service call only when an *event* occurs. The definition of

events is application dependent. Whenever an event occurs in the server side, the application should call a special Web service that indicates that the event occurred, eventually waking up the continuous service to reevaluate the service for all its clients. In the dQSQ system, we defined an event to be the receiving of new data by the server. This makes sense, since whenever a server receives new data from an external source, it may also produce new data for the clients of this server. Using Event-Sensitive continuous services achieved a big improvement in the performance and robustness of both the AXML framework, and in particular the dQSQ system.

- **Dynamic AXML repositories**. By default, the AXML repositories of documents and services are defined in advance, before the AXML peer is started. Then, when the AXML peer is initialized, it loads the documents and services in the repositories. Such a flow is not practical to dQSQ. Since the dQSQ system generates the rewritten program dynamically, while AXML is already running, we needed a way to add new AXML documents and services on the fly. Similarly, since the dQSQ system is an interactive system that enables the user to run sequential queries, we needed a way to remove AXML documents and services when the current query computation is finished. Therefore we added to AXML Web services to support the missing operations, e.g. the services *addDocument* and *addService*.

- **Continuous services usability**. As the dQSQ system is, as far as we know, the first real application that ever used continuous services, we made several bug fixes that actually turned continuous

services to a usable mechanism.

- **Various bug fixes**. A few bug fixes that were required for the normal functioning of the system. This included fixing a deadlock between threads and dealing with unhandled exceptions.