The Webdamlog System

Managing Distributed Knowledge on the Web [*]

Serge Abiteboul
Inria Saclay & ENS Cachan
France
first.last@inria.fr

Émilien Antoine
Inria Saclay & ENS Cachan
France
first.last@inria.fr

Julia Stoyanovich
Drexel University, USA
Skoltech, Russia
stoyanovich@drexel.edu

April 16, 2013

We study the use of *WebdamLog*, a declarative high-level language in the style of datalog, to support the distribution of both *data* and *knowledge* (i.e., programs) over a network of autonomous peers. The main novelty of *WebdamLog* compared to datalog is its use of delegation, that is, the ability for a peer to communicate a program to another peer.

We present results of a user study, showing that users can write *WebdamLog* programs quickly and correctly, and with a minimal amount of training. We present an implementation of the *WebdamLog* inference engine relying on the *Bud* datalog engine. We describe an experimental evaluation of the *WebdamLog* engine, demonstrating that *WebdamLog* can be implemented efficiently. We conclude with a discussion of ongoing and future work.

1 Introduction

A number of works have argued for developing a holistic approach to distributed content management, e.g. *P2P Content Warehouse* [1], *Dataspaces* [11] and *Data rings* [6]. The goal is to facilitate the collaboration of autonomous peers towards solving content management tasks. Such situations arise for instance in personal information management (PIM), which is often given as an important motivating example [11]. In [6], the authors argued for founding such data exchange on declarative languages, to facilitate the design of applications, notably by non-technical users.

In the present work, we propose an approach for managing data and knowledge (i.e., programs) over a network of autonomous peers. From a system viewpoint, the different actors are autonomous and heterogeneous in the style of P2P [6, 11]. However, we do not see the system we developed as an alternative to existing network services such as Facebook or Flickr. Rather, we view our system as the means of seamlessly managing distributed knowledge residing in any of these services, as well as in a wide variety of systems managing personal or social data.

Our system uses the *WebdamLog* language [4], a declarative high-level language in the style of datalog, to support the distribution of both *data* and *knowledge* (i.e., programs) over a network of autonomous peers. In recent years, there has been renewed interest in using languages in the datalog family in a large range of applications, from program analysis, to security and privacy protocols, to natural language processing, to multi-player games. The arguments in favor of datalog-style languages are familiar ones: a declarative approach alleviates the conceptual complexity on the user, while at the same time allowing for powerful performance optimizations on the part of the system.

WebdamLog is a datalog-style language that emphasizes cooperation between autonomous peers communicating in an asynchronous manner. The *WebdamLog* language extends datalog in a number of ways, supporting updates [9], distribution [3], negation [12], and, importantly, a novel feature called delegation [4]. As a result, *WebdamLog* is neither as simple nor as beautiful as datalog. It is also more procedural, which is needed to capture real Web applications with the peers' knowledge evolving over time.

We illustrate by example (Section 2) that the language (formally recalled in Section 3) is indeed well adapted to specifying realistic distributed content management tasks, notably in PIM. Our technical contributions are described in the following sections:

- We present results of a user study, showing that users can write *WebdamLog* programs quickly and correctly, and with a minimal amount of training (Section 4).
- We present an implementation of the *WebdamLog* engine relying on the *Bud* datalog engine (Section 5). Our implementation supports novel linguistic features such as peer and predicate variables and rule delegation.
- We describe an experimental evaluation of the *WebdamLog* engine (Section 6).

We discuss related work in Section 7, outline future research directions and conclude in Section 8.

2 Running example

Suppose that Alice and Bob are getting married, and their friends want to offer them an album of photos in which the bride and groom appear together. Such photos may be owned by friends and family members of Alice and Bob. Owners of the photos may store them on a variety of services and devices, including, e.g., desktop computers, smartphones, Picasa, and Flickr.

Making a photo album for Alice and Bob involves the following steps: (1) Identify friends of Alice and Bob using Facebook and Google+; (2) Find out where each friend keeps his/her photos and how to access them; (3) From among all photos that are obtained, select those that feature both Alice and Bob, using, e.g., tags or face recognition software; and (4) Ask Sue, a friend of Alice, to verify that the selected photos are appropriate for the photo album and to possibly exclude some from this album.

As should be clear from the example, such a task would be much more manageable if it were executed automatically. Its execution involves a certain amount of simple reasoning on the

[*]This work has been partially funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013); ERC grantWebdam, agreement 226513. `http://webdam.inria.fr/`

part of the system, which can be naturally specified with declarative rules. For example, for Step (1), the following *WebdamLog* rule computes the union of Alice's and Bob's Facebook contacts in a relation allFriends on Sue's peer:

```
[rule at sue]
allFriends@sue($name) :- friends@aliceFB($name)
allFriends@sue($name) :- friends@bobFB($name)
```

using wrappers to Facebook for Alice and Bob.

In general, a *peer name* such as aliceFB or sueIPhone denotes a system or a device associated to a particular URL. Also a *relation name* such as friends or contacts denotes the name of a relation or a service on the corresponding system/device.

For simplicity, we assume that a person's name, e.g. alice, corresponds to the name of the *peer* that the particular friend uses as entry point to the *Webdam* system. (This name is thus associated to a particular URL.) We assume that each such peer keeps localization data for the corresponding person. For instance, relation photoLocation in that peer tells where (i.e., at which peers) this person keeps her photos. The following rule, at peer sue, *delegates* Steps (2) and (3) of the photo album task to the peers corresponding to the peers corresponding to her friends:

```
[rule at sue]
album@sue($photo,$name) :-
   allFriends@sue($name),
   photoLocation@$name($peer),
   photos@$peer($photo),
   features@$peer($photo,alice),
   features@$peer($photo,bob)
```

The key feature of this rule is the use of the *WebdamLog* language to share the work. Let Dan be a friend, and so a possible source. Then Sue's peer will delegate the following rule to Dan's peer:

```
[rule at dan]
album@sue($photo,dan) :-
   photoLocation@dan($peer),
   photos@$peer($photo),
   features@$peer($photo,alice),
   features@$peer($photo,bob)
```

Now suppose that Dan uses both Picasa and Flickr. Then, Dan's peer will delegate to danPicasa (a wrapper for Dan's account on Picasa) the following rule:

```
[rule at danPicasa]
album@sue($photo,dan) :-
   photos@danPicasa($photo),
   features@danPicasa($photo,alice),
   features@danPicasa($photo,bob)
```

and similarly for Flickr.

Note how the tasks are automatically shared by many peers. Observe that when new friends of Alice or Bob are discovered (e.g., proposed by some known friends), Sue's album, which is defined *intentionally*, is automatically updated. Observe also that, to simplify, we assume here that all peers use a similar organization (ontology). This constraint may easily be removed at the cost of slightly more complicated rules.

Now consider Step (4) in the photo album task. Sue may decide, for instance, that photos of the couple from Dave's Flickr stream are inappropriate, and that Dave should be excluded from the set of sources. Such manual curation by Sue may be accomplished by modifying the definition of allFriends:

```
[rule at sue]
allFriends@sue($name) :- friends@aliceFB($name),
                         not blocked@sue($name)
allFriends@sue($name) :- friends@bobFB($name)
                         not blocked@sue($name)
```

By inserting/removing facts in blocked@sue, Sue now controls who can participate. A similar control can also be added at the photo or photo location level.

Observe that updates result in modifying the programs running at the participating peers. For instance, the sets of rules at the various peers evolve, controlled by Sue's updates as well as by the discovery of new friends of Alice or Bob, and of new sources of photos. Consequently, the album evolves as well.

We will use the example of this section throughout the paper to demonstrate the salient features of our approach.

3 The WebdamLog Language

In this section, we briefly recall the language *WebdamLog* from [4].

We assume the existence of a countable set of variables and of a countable set of data values that includes a set of relation names and a set of peer names. (Relation and peer names are part of the data.) Variables start with the symbol \$, e.g. $\$x$.

Schema. A *relation* in our context is an expression $m@p$ where m is a relation name and p a peer name. A *schema* is an expression (π, E, I, σ) where π is a possibly infinite set of peer names, E is a set of extensional relations of the form $m@p$ for $p \in \pi$, I is a set of intentional relations of the form $m@p$ for $p \in \pi$, and σ, the sorting function, specifies for each relation $m@p$, an integer $\sigma(m@p)$ that is its sort. A relation cannot be at the same time intentional and extensional.

Facts. A *fact* is an expression of the form $m@p(a_1, ..., a_n)$, where $n = \sigma(m@p)$ and $a_1, ..., a_n$ are data values. An example of a fact is:
pictures@myalbum(1771.jpg, "“Timbuktu′′", 11/11/2011)

Rules. A *term* is a constant or a variable. A *rule* in a peer p is an expression of the form:

$$\text{[at p]}\quad \$R@\$P(\$U){:}\text{-}(\neg)\ \$R_1@\$P_1(\$U_1),\ldots,\\ (\neg)\ \$R_n@\$P_n(\$U_n)$$

where \$R, $\$R_i$ are relation terms, \$P, $\$P_i$ are peer terms, \$U, $\$U_i$ are vectors of terms. The following safety condition is imposed: that \$R and \$P must appear positively bound in the body and each variable occurring in a negative literal must also appear positively bound in the body. In addition, rules are required from left to right and it is also required that each peer name $\$P_i$ must be positively bound in a previous atom.

Semantics. At a particular point in time, each peer p has a *state* consisting of some facts, some rules specified locally, and possibly of some rules that have been delegated to p by other peers. Peers evolve by updating their base of facts, by sending facts to other peers, and by updating their delegations to other peers. So, both the set of facts and the set of delegated rules evolve over time. (To simplify, we follow [4] in assuming that the set of rules specified locally is fixed.)

The semantics of a rule with head m@p(u) in a peer p′ depends on the nature of the relation in its head: whether it is extensional (m@p in E) or intentional (m@p in I), and whether it is local (p=p′) or not. We first consider rules in which all relations occurring in the body are local; we call such rules *local rules*. A subtlety lies in the use of variables for peer names. The nature of a rule may depend on the instantiation of its variables, i.e., one instantiation of a particular rule may be local, whereas another may not be.

We distinguish 5 cases identified by a letter in which we classify the rules.

A. Local rule with local intentional head (datalog) These rules define local intentional predicates, as in classic datalog.

B. Local rule with local extensional head (local database updates) Facts derived by this kind of rules are inserted into the local database. Note that, by default, like in Dedalus[9], facts are not persistent. To have them persist, we use rules of the form m@p(U) :- m@p(U). Deletion can be captured by controlling the persistence of facts.

The two previous kinds of rules, containing only predicates of the local peer, do not require network communication, and are not affected by problems due to asynchronicity of the network.

C. Local rule with non-local extensional head (messaging) Facts derived by rules of this kind are sent to other peers. For example, the rule:

```
[at mi] $m@$p($name, ""Happy birthday!"") :-
        today@mi($date),
        birthday@mi($name, $m, $p, $date)
```

where mi stands for my iPhone, results in sending a Happy Birthday message to a contact on the day of his birthday. Observe that the name $p of the peer and the name $m of the message varies depending on the person.

D. Local rule with non-local intentional head (view delegation) Such a rule results in installing a view remotely. For instance, the rule

```
[at mi] boyMeetsGirl@gossipsite($girl, $boy) :-
        girls@mi($girl, $loc),
        boys@mi($boy, $loc)
```

installs a join of two mi relations at gossipsite.

Finally we consider non-local rules.

E. Non-local (general delegation) Consider the rule

```
[at mi] boyMeetsGirl@gossipsite($girl, $boy) :-
        girls@mi($girl, $loc), boys@ai($boy, $loc)
```

where ai stands for Alice's iPhone. This results in installing, at gossipsite, a view $t_{r@mi}$ and a rule, defined as follows:

```
[at mi] t_r@mi@ai($girl, $loc) :-
        girls@mi($girl, $loc)
[at ai] boyMeetsGirl@gossipsite($girl, $boy) :-
        t_r@mi@ai($girl, $loc), boys@ai($boy, $loc)
```

Note that both rules are now local. Note also that, when girls@mi changes, this modifies the view at Alice's iPhone, possibly changing the semantics of boyMeetsGirl@gossipsite.

In [4], we formally define the semantics of *WebdamLog*. We show that, unless all peers and programs are known in advance, delegation strictly increases the expressive power of the model. If they are known in advance, delegation does not bring any extra power. Of course, delegation is also useful in practice, because it enables obtaining logic (rules) from other sites, and deploying logic (rules) to other sites. Conditions for systems to be deterministic are shown in [4], and are extremely restrictive. Even in the absence of negation, a *WebdamLog* system will typically not be deterministic because of asynchronicity.

4 Usability of WebdamLog

We argued in the introduction that *WebdamLog* can be used to declaratively specify distributed tasks in a variety of applications, including personal data management. We conducted a user study to demonstrate the usability of *WebdamLog* in this particular domain.

Participants. We recruited 27 participants for the user study. We present a break-down of results by two groups.

Group 1 consisted of 16 participants with training in Computer Science. Among them, 5 had basic database background, and 4 were familiar with advanced database concepts, including datalog. The group had the following break-down by highest completed education level: 2 high school, 3 BS, 9 MS, and 2 PhD.

Group 2 consisted of 11 participants with no CS training, and with the following break-down by highest completed education level: 3 vocational school, 6 BS, 2 MS.

Study design. All participants were given a brief tutorial in which basic features of *WebdamLog* were explained informally, and demonstrated through examples. The tutorial took 15-20 minutes for *Group 1* and 25 minutes for *Group 2*. Following the tutorial, all participants were asked to take a written test. The test consisted of three problems that tested comprehension of different features of *WebdamLog*, including local and non-local rules, rules with variable relation and peer names, and delegation. In the tutorial and the test, we did not make an explicit distinction between intentional and extensional relations, and we ignored recursion.

The user study test had the following contents, reproduced here literally, apart from formatting.

Problem 1. Consider the following relations and facts.

```
schema: songs(fileName,content) // the same at all peers

songs@lastFM("song1.mp3", "...")
songs@lastFM("song2.mp3", "...")
songs@lastFM("song3.mp3", "...")
songs@pandora("song4.mp3", "...")
songs@pandora("song5.mp3", "...")
```

1. Write one or several rules that copy all songs from `lastFM` and `Pandora` into relation `songs` at peer `myLaptop`.

2. Suppose now that relation `peers@myLaptop` contains names of peers on which to look for music. You can assume that each peer stores songs in a relation called `songs`, with the same schema as above. Write a *WebdamLog* program that copies songs from all peers into `songs@myLaptop`.

3. Write a rule that copies songs from `songs@myLaptop` into the `songs` relation on each peer whose name is listed in `peers@myLaptop`.

Problem 2. Consider the following relations and facts.

```
schema: friends(friendName)  photos(fileName,content)
          inPhoto(fileName, friendName)

friends@facebook("ann")
friends@facebook("sue")
friends@facebook("zoe")

photos@ann("sunset.jpg", "...")
photos@ann("vacation.jpg", "...")
photos@ann("party.jpg", "...")

photos@sue("image1.jpg","...")
photos@sue("image2.jpg","...")

inPhoto@ann("vacation.jpg", "jane")
inPhoto@ann("vacation.jpg", "ann")
inPhoto@ann("party.jpg", "jane")
```

```
inPhoto@ann("party.jpg", "zoe")
inPhoto@ann("party.jpg", "sue")

inPhoto@sue("image2.jpg", "sue")
inPhoto@sue("image2.jpg", "jane")
```

Assume that photos and inPhoto relations at all peers have the same schema. Consider now the following *WebdamLog* rule.

```
photos@myLaptop($X,$Z) :- friends@facebook($Y),
          photos@$Y($X,$Z), inPhoto@$Y($X,"jane")
```

1. Explain in words what this rule computes.

2. List the facts in that are in photos@myLaptop after the rule above is executed.

3. List the facts that are in photos@myLaptop if the following rule is executed instead:

```
photos@myLaptop($X,$Z) :- friends@facebook($Y),
          photos@$Y($X,$Z), inPhoto@$Y($X,"jane"),
          inPhoto@$Y($X,"sue")
```

Problem 3. Recall the example from the tutorial, in which we looked at subscribing the peer myLaptop to CNN news. This example is reproduced below.

```
schema: news@cnn(text)    news@myLaptop(source, text)
        subscribers@cnn(peer)

news@cnn("US Olympic gold")
news@cnn("Higgs boson seen in action")
subscribers@cnn("myLaptop")

[at cnn] news@$X("cnn", $Y) :- subscribers@cnn($X),
                               news@cnn($Y)
```

Suppose that you would now like to receive CNN news on peer myPhone, and to store them in relation news, with the schema source,text. Describe at least 1 method for doing this. You may assume that you can add rules at peers cnn, myLaptop and myPhone, and that you can insert facts into relations on any of these peers.

Results. The results of the study were very encouraging.

Group 1. On Problem 1, 3 participants received a score of 2.5 out of 3, while 13 participants received a perfect score. All participants received a perfect score on Problem 2. Problem 3 was open-ended, and all participants gave at least one correct answer. 4 participants gave 3 correct answers, 4 gave 2 correct answers (2 of these also gave 1 incorrect answer each), and the remaining 8 participants each gave 1 correct answer.

We also asked participants to record how long it took them to answer each problem, in minutes. Problem 1 took between 2.5 and 6 minutes, Problem 2 between 2 and 9 minutes, and Problem 3 between 1 and 8 minutes. We did not observe any correlation between the time it took to answer questions and the participant background in data management or even datalog.

Group 2. On Problem 1, the average score was 2.3, with the following break-down: 6 participants received a perfect score, 3 received 2 out of 3, 1 had a score of 1, and 2 were not able to solve the problem. On Problem 2, 10 participants received a perfect score and 1 got a score of 2 out of 3. On Problem 3, 1 gave 5 good answers, 6 gave 3 good answers, 3 gave 2 good answers, and 2 gave no correct answer. The same two participants failed to answer Problems 1 and 3.

The test took longer for the participants without CS training. Problem 1 took between 6 and 8 minutes to solve in this group, Problem 2 took between 5 and 8 minutes, and Problem 3 took between 4 and 12 minutes.

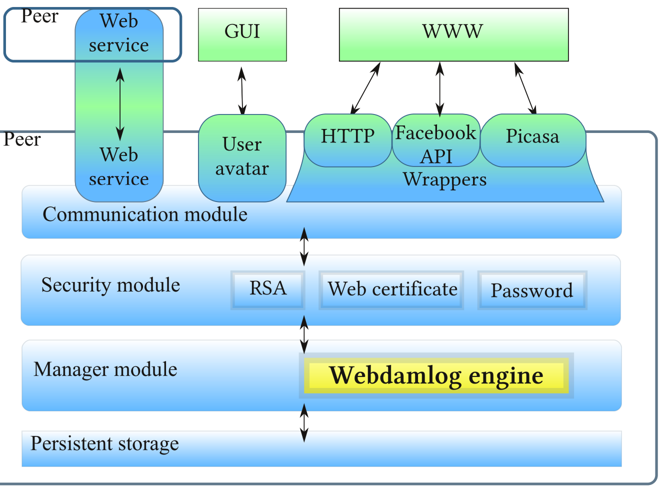

Figure 1: *WebdamLog* engine in a full *Webdam* peer

Remark. We considered alternative ways in which a user can interact with a *WebdamLog* system. We are currently developing an interface in which users will be able to write *WebdamLog* programs, but will also have access to customizable canned queries implementing common functionality. A SQL-based approach is not a natural choice, since SQL does not accommodate distribution, which is central to *WebdamLog*.

In summary, all technical and the majority of non-technical participants of our study were able to both understand and write *WebdamLog* programs correctly, with a minimal amount of training. We observed a difference between the technical and non-technical groups in terms of both correctness and time to solution. Two members of the non-technical group were able to understand *WebdamLog* programs but were not able to write programs on their own. We believe that this issue will be alleviated once an appropriate user interface becomes available.

5 The WebdamLog System

In this section, we describe the architecture of the *WebdamLog* system. We describe the implementation of the system, stressing the novel features compared to standard datalog engines.

5.1 System architecture

Figure 1 shows the architecture of a *WebdamLog* peer. Facts and rules are stored in a persistent store. The *WebdamLog* engine, described in greater detail in the remainder of this section, retrieves these facts and rules to process updates and answer queries coming from the top layers. The Security module provides facilities for standard access control mechanisms such as encryption, signatures and other authentication protocols. The Communication module is responsible for exchanging facts and rules with other peers.

Datalog evaluation has been intensively studied, and several open-source implementations are available. We chose not to implement yet another datalog engine, but instead to extend an existing one. In particular, we considered two open-source systems that are currently being supported, namely, *Bud* [21] from Berkeley University and *IRIS* [19] from Innsbruck University. The *IRIS* system is implemented in Java and supports the main strategies for efficient evaluation of standard local datalog. The *Bud* system is implemented in the Ruby scripting language, and initially seemed less promising from a performance viewpoint. However, *Bud* provides mechanisms for asynchronous communication between peers, an essential feature for *WebdamLog*.

In absence of a real performance comparison, the choice was not easy. We finally decided in favor of *Bud*, both because of its support for asynchronous communication, and because its scalability has been demonstrated in real-life scenarios such as Internet routing.

5.2 WebdamLog computation in Bud

The *Bud* system supports a powerful datalog-like language introduced in [8]. Indeed, we see *Bud* (and use it) as a distributed datalog engine with updates and asynchronous communications.

A *WebdamLog* computation consists semantically of a sequence of *stages*, with each stage involving a single peer. Each stage of a *WebdamLog* peer computation is in turn performed by a three-step *Bud* computation, described next. Note that we use the word *stage* for *WebdamLog* and *step* for *Bud*:

$$\dots \left| \begin{array}{ccc} \multicolumn{3}{c}{\text{Stage at peer p}} \\ \text{Step 1} & \text{Step 2} & \text{Step 3} \end{array} \right| \begin{array}{ccc} \multicolumn{3}{c}{\text{Stage at peer q}} \\ \text{Step 1} & \text{Step 2} & \text{Step 3} \end{array} \right| \dots \tag{1}$$

The 3 steps of a *WebdamLog* stage are as follows:

1. Inputs are collected including input messages from other peers, clock interrupts and host language calls.

2. Time is frozen; the union of the local store and of the batch of events received since the last stage is taken as an extensional database, and a *Bud* program is run to fixpoint.

3. Outputs are obtained as side effects of the program, including output messages to other peers, updates to the local store, and host language callbacks.

Observe that a fixpoint computation is performed at Step 2 by the local datalog engine (namely the *Bud* engine). This computation is based on a fixed program with no deletion over a fixed set of extensional relations. In Step 3, deletion messages may be produced, along with updates to the set of rules and to the set of extensional relations (for different reasons, which we will explain further). Note that all this occurs *outside* the datalog fixpoint computation.

Relations appearing in the rules are implemented as *Bud* collections. *Bud* distinguishes between three kinds of key-value sets:

1. A *table* keeps a fact until an explicit delete order is received. We use tables to support *WebdamLog* extensional relations.

2. A *scratch* is used for storing results of intermediate computation. We use *scratch* collections to implement *WebdamLog* local intentional relations. It is emptied at Step 1 and receives facts during fixpoint computation at Step 2.

3. A *channel* provides support for asynchronous communications. It records facts that have to be sent to other peers. We use channels for that and in particular for messages related to installing or removing delegations.

As in *WebdamLog*, facts in a peer are consumed by the engine at each firing of the peer (each stage). To make facts persistent, they have to be re-derived by the peer at each stage. This is captured in our implementation by assuming that rules rederive extensional facts implicitly, unless a deletion message has been received.

We observe a subtle point that lead us to not fully adopt the original semantics of *WebdamLog*, as described in [4]. There, we assumed for simplicity that messages are transmitted instantaneously. This assumption is not realistic in practice, and does not hold in our implementation. Since communications are asynchronous, there is no guarantee in *WebdamLog* as to when a fact written to a channel will be received by a remote peer.

5.3 Implementing WebdamLog rules

We now describe how *WebdamLog* rules are implemented on top of *Bud*. We distinguish between 4 cases. This brings us to revisit the semantics of *WebdamLog* (from Section 3) with a focus on implementation. As in Section 3, whether a rule in a peer p is *local* (i.e., all relations occurring in the rule body are p-relations) plays an important role. We consider 4 cases. The last case (Case F) focuses on the use of variables for relation and peer names. For the first 3 cases, we ignore such variables.

A-B-C. Simple local rules. In this cases, i.e., local rules with either an extensional relation or a local intentional relation in the head, *WebdamLog* rules can be directly supported by identical *Bud* rules. (This takes care of local deduction as in datalog (A), messages for local updates (B) and messages to other peers (C).)

D. Local with non-local intentional head. From an implementation viewpoint, this case is more tricky. We illustrate it with an example. Consider an intentional relation $\mathsf{s_0@q}$ defined in the distributed setting by the following two rules:

[at p1] $\mathsf{s_0@q(X,Y)\ \text{:-}\ r_1@p_1(X,Y)}$
[at p2] $\mathsf{s_0@q(X,Y)\ \text{:-}\ r_1@p_2(X,Y)}$

Intuitively, the two rules specify a view relation $\mathsf{s_0@q}$ at q that is the union of two relations $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$ from peers $\mathsf{p_1}$ and $\mathsf{p_2}$, respectively. Consider a possible naive implementation that would consist in materializing relation s_0 at q, and having $\mathsf{p_1}$ and $\mathsf{p_2}$ send update messages to q. Now suppose that a tuple $\langle 0,1\rangle$ is in both $\mathsf{r_1@p_1}$ and $\mathsf{r_1@p_2}$. Then it is correctly in $\mathsf{s_0@q}$. Now suppose that this tuple is deleted from $\mathsf{r_1@p_1}$. Then a deletion message is sent to q, resulting in wrongly deleting the fact from $\mathsf{s_0@q}$.

The problem arises because the tuple $\langle 0,1\rangle$ originally had two reasons to be in s_0, and only one of the reasons disappeared. To avoid this problem, we could use the *provenance* of the fact $\langle 0,1\rangle$ in $\mathsf{s_0@q}$.

A general approach for tracking provenance in our setting, and to using it as basis for performance optimizations, is part of ongoing work, and is outlined in Section 5.5. For now, we can implement the following *Bud* rules at $\mathsf{p_1}$, $\mathsf{p_2}$ to correctly support the two rules:

[at p1] $\mathsf{s_{0p1}@q(X,Y)\text{:-}\ r_1@p_1(X,Y)}$
[at p2] $\mathsf{s_{0p2}@q(X,Y)\text{:-}\ r_1@p_2(X,Y)}$
[at q] $\mathsf{s_0@q(X,Y)\text{:-}\ s_{0p1}@q(X,Y)}$
[at q] $\mathsf{s_0@q(X,Y)\text{:-}\ s_{0p2}@q(X,Y)}$

Note that relations $\mathsf{s_{0p1}}$ and $\mathsf{s_{0p2}}$ may be either intentional, in which case the view is computed on demand, or extensional, in which case the view is materialized.

E. Non-local rules. We consider non-local rules with extensional head. (Non-local rules with intentional head are treated similarly.) An example of such a rule is:

[at p] $\mathsf{r_0@q(\overline{X_0})\text{:-}\ r_1@q_1(\overline{X_1}),\dots,r_i@q_i(\overline{X_i}),\dots}$

with $\mathsf{q_1}= \dots = \mathsf{q_{i-1}} = \mathsf{p}$, $\mathsf{q_i} = \mathsf{q} \neq \mathsf{p}$, and with each $\overline{\mathsf{X_j}}$ denoting a tuple of terms. If we consider atoms in the body from left to right, we can process at p the rule until we reach $\mathsf{r_i@q(\overline{X_i})}$. Peer p does not know how to evaluate this atom, but it knows that the atom is in the realm of q. Therefore, peer p rewrites the rule into two rules, as specified by the formal definition of delegation in *WebdamLog*:

[at p] $\mathsf{mid@q}(\overline{\mathsf{X_{mid}}})$:- $\mathsf{r_1@p}(\overline{\mathsf{X_1}})$,...,$\mathsf{r_{i-1}@p}(\overline{\mathsf{X_{i-1}}})$
[at q] $\mathsf{r_0@q}(\overline{\mathsf{X_0}})$:- $\mathsf{mid@q}(\overline{\mathsf{X_{mid}}})$, $\mathsf{r_i@q}(\overline{\mathsf{X_i}})$,...

where *mid* identifies the message, and notably encodes, (i) the identifier of the original rule, (ii) that the rule was delegated by p to q, and (iii) the split position in the original rule. The tuple $\overline{\mathsf{X_{mid}}}$ includes the variables that are needed for the evaluation of the second part of the rule, or for the head. Observe that the first rule (at p) is now local. If the second rule, installed at q, is also local, no further rewriting is needed. Otherwise, a new rewriting happens, again splitting the rule at q, delegating the second part of the rule as appropriate, and so on.

Observe that an evolution of the state of p may result in installing new rules at q, or in removing some delegations. Deletion of a delegation is simply captured by updating the predicate guarding the rule. Insertion of a new delegation modifies the program at q. Note that in *Bud* the program of a peer is fixed, and so adding and removing delegations is a novel feature in *WebdamLog*. Implementing this feature requires us to modify the *Bud* program of a peer. This happens during Step 1 of the *WebdamLog* stage.

F. Relation and peer variables. Finally, we consider relation and peer variables. In all cases presented so far, *WebdamLog* rules could be compiled statically into *Bud* rules. This is no longer possible in this last case. To see this, consider an atom in the body of a rule. Observe that, if the peer name in this atom is a variable, then the system cannot tell before the variable is instantiated whether the rule is local or not. Also, observe that, if the relation name in this atom is a variable, then the system cannot know whether that relation already exists or not. In general, we cannot compile a *WebdamLog* rule into *Bud* until all peer and relation variables are instantiated.

To illustrate this situation more precisely, consider a rule of the form:

$\mathsf{r_0@p}(\overline{\mathsf{X_0}})$:- $\mathsf{r_1@p}$($X), ...,$X@p($\overline{\mathsf{X_i}}$),...,

where $\mathsf{r_0@p}$ is extensional and $X is a variable. This particular rule is relatively simple since, no matter how the variable is instantiated, the rule falls into the simple case **B**. However, it is not a *Bud* rule because of the variable relation name $X.

Note that *WebdamLog* rules are evaluated from left to right, and a constraint is that each relation and peer variable must be bound in a previous atom. (This constraint is imposed by the language.) Therefore, when we reach the atom $X@p($\overline{\mathsf{X_i}}$), the variable $X has been instantiated.

To evaluate this rule, we use two *WebdamLog* stages of the peer. In the first stage, we bind $X with values found by instantiating $\mathsf{r_1@p}$($X). Suppose that we find two values for $X, say t_1 and t_2. We always wait for the next stage to introduce new rules (there are two new rules in this case). More precisely, new rules are introduced during Step 1 of the *WebdamLog* computation of the next stage. In the example, the following rules are added to the *Bud* program at *p*:

$\mathsf{r_0@p}(\overline{\mathsf{X_0}})$:- t1@p($\overline{\mathsf{X_i}}$),...,
$\mathsf{r_0@p}(\overline{\mathsf{X_0}})$:- t2@p($\overline{\mathsf{X_i}}$),...,

Observe that, even in the absence of delegation, having variable relation and peer names allows the *WebdamLog* engine to produce new rules at run time, possibly leading to the creation of new relations. This is a distinguishing feature of our approach, and is novel to *WebdamLog* and to our implementation.

This example uses a relation name variable. Peer name variables are treated similarly. Observe that having a peer name variable, and instantiating it to thousands of peer names, allows us delegating a rule to thousands of peers. This makes distributing computation very easy from the point of view of the user, but also underscores the need for powerful security mechanisms. Developing such mechanisms is in our immediate plans for future work.

5.4 Running the fixpoint

The *Bud* engine evaluates the fixpoint using the semi-naive algorithm, i.e., *Bud* saturates one stratum after another according to a stratification given by the *dependency graph.* The dependency graph is a directed hyper-graph with relations as nodes, and with a hyper-edge from relations s_i to relation r if there is a rule in which all s_i appear in the body and r appears in the head. Since this is classic material, we omit the details but observe that, since *WebdamLog* rules may be added or removed at run-time, the program evolves, leading to changes in the dependency graph. Therefore, the dependency graph is recomputed at step 1 of a *WebdamLog* stage when receiving new rules, and remains fixed for the following step 2. The *WebdamLog* engine pushes further the differentiation technique that serves as basis of the semi-naive algorithm.

Although, according to *WebdamLog* semantics, facts are consumed and possibly re-derived, it would be inefficient to recompute the proof of existence of all facts at each stage. Between two consecutive stages, each relation keeps a cache of its previous contents. This cache may be invalidated by *WebdamLog* if a newly installed rule creates a new dependency for this relation. Note that *Bud* already performs cache invalidation propagation for facts, which we adapt to fit *WebdamLog* semantics. This incremental optimization across stages allows us to run the fixpoint computation only on the relations that may have changed since the previous stage.

5.5 Maintaining dynamic peer state

A *WebdamLog* system executes in a highly dynamic environment, where peer state frequently changes, in terms of both data and program, and where peers may come and go. This is a strong departure from datalog-based systems such as *Bud* that assume the set of peers and rules to be fixed. As part of our ongoing work, we are focusing on efficiently supporting dynamic changes in peer state, with the help of a novel kind of a *provenance graph.*

We use provenance graphs to record the derivations of *WebdamLog* facts and rules, and to capture fine-grained dependencies between facts, rules, and peers. We build on the formalism proposed in [13], where each tuple in the database is annotated with an element of a provenance semiring, and annotations are propagated through query evaluation. Provenance can be used for a number of purposes such as explaining query results or system behavior, and for debugging. Our primary use of provenance is to optimize performance of *WebdamLog* evaluation in presence of deletions. We are also currently investigating the use of provenance for enforcing *access control* and for detecting access control violations.

6 Experimental Evaluation

The goal of the experimental evaluation is to verify that *WebdamLog* programs can be executed efficiently. We show here that rewriting and delegation can be implemented efficiently.

In the experiments, we used synthetically generated data. All experiments were conducted on up to 50 Amazon EC2 micro instances, with 2 *WebdamLog* peers per instance. Micro-instances are virtual machines with two process units, Intel(R) Xeon(R) CPU E5507 @2.27GHz with 613 MB of RAM, running Ubuntu server 12.04 (64-bit). All experiments were executed 4 times with a warm start. We report averages over 4 executions.

The cost of delegation. We now focus is on measuring *WebdamLog* overhead in dealing with delegations. Recall the *Bud* steps performed by each peer at each *WebdamLog* stage, described in Section 5.2. We can break down each step into *WebdamLog*-specific and *Bud*-specific tasks as follows:

1. Inputs are collected
   a) ***Bud*** reads the input from the network and populates its channels.
   b) ***WebdamLog*** parses the input in channels and updates the dependency graph with new rules. The dependency graph is used to control the rules that are used in the semi-naive evaluation (see Section 5.4).
2. Time is frozen
   a) ***Bud*** invalidates each Δ (used by the semi-naive evaluation) that has to be reevaluated because it corresponds to a relation that may have changed.
   b) ***WebdamLog*** invalidates Δ according to program updates. Moreover, *WebdamLog* propagates deletions. (Recall that the semi-naive evaluation deals only with tuple additions.)
   c) ***Bud*** performs semi-naive fixpoint evaluation for all invalidated relations, taking the last Δ for differentiation.
3. Outputs are obtained
   a) ***WebdamLog*** builds packets of rules and updates to send.
   b) ***Bud*** sends packets.

We report the running time of *WebdamLog* as the sum of Steps 1b, 2b and 3a, and the running time of *Bud* as the sum of Steps 1a, 2a, 2c and 3b. All running times are expressed in percentage of the total running time, which is measured in seconds. For each experiment, we will see that the running time of *WebdamLog*-specific phases is reasonable compared to the overall running time.

Non-local rules. In the first experiment, we evaluate the running time of a non-local rule with an extensional head. Rules of this kind lead to delegations. We use the following rule:

```
[at alice]
join@sue($Z) :- rel1@alice($X,$Y), rel2@bob($Y,$Z)
```

This rule computes the join of two relations at distinct peers (rel1@alice and rel2@bob), and then installs the result, projected on the last column, at the third peer (join@sue). Relations rel1@alice and rel2@bob each contain 1 000 tuples that are pairs of integers, with values drawn uniformly at random from the 1 to 100 range. In the next table, we report the total running time of the program at each peer, as well as the break-down of the time into *Bud* and *WebdamLog*.

	WebdamLog	*Bud*	total
alice	10.8%	89.2%	0.10s
bob	4.0%	96.0%	0.87s
sue	0.7%	99.3%	0.02s

The portion of the overall time spent on *WebdamLog* computation on alice is fairly high: 10.8%. This is because that peer's work is essentially to delegate the join to bob. Peer bob spends most of its time computing the join, a *Bud* computation. Peer sue has little to do. As can be seen from these numbers, the overhead of delegation is small.

Relation and peer variables. In the second experiment, we evaluate the execution time of a *WebdamLog* program for the distributed computation of a union. The following rule uses relation and peer variables and executes at peer sue:

```
[at sue]
union@sue($X) :- peers@sue($Y,$Z), $Y@$Z($X)
```

The relation peers@sue contains 12 tuples corresponding to 3 peers (including sue) with 4 relations per peer. Thus, the rule specifies a union of 12 relations. Each relation participating in the union contains 1 000 tuples, each with a single integer column, and with values for the attribute drawn independently at random between 1 and 10 000.

	WebdamLog	*Bud*	total
sue	9.9%	90.1%	1.04s
remote1	1.1%	98.9%	0.04s
remote2	1.3%	98.7%	0.04s

Observe that sue does most of the work, both delegating rules and also computing the union. The *WebdamLog* overhead is 9.9%, which is still reasonable. The running time on remote peers is very small, and the *WebdamLog* portion of the computation is negligible.

QSQ-style optimization. In this experiment, we measure the effectiveness of an optimization that can be viewed as a distributed version of query subquery (QSQ) [22], where only the relevant data are communicated at query time. More precisely, we consider the following view union2 on peer sue, defined as the union of two relations.

```
[at sue]
union2@sue($name,$X) :- friendPhotos@alice($name,$X)
union2@sue($name,$X) :- friendPhotos@bob($name,$X)
```

Suppose we want to obtain the photos of Charlie, i.e. the tuples in union2 that have the value "Charlie" for first attribute. We vary the number of facts in friendPhotos@alice and friendPhoto@bob that match the query. We compare the cost of materializing the entire view to answer the query to that of installing only the necessary delegations computed at query time to compute the answer.

Results of this experiment are presented in Figure 2. We report the waiting time at sue. As expected, QSQ-style optimization brings important performance improvements (except when almost all facts are selected). This shows its usefulness in such a distributed setting.

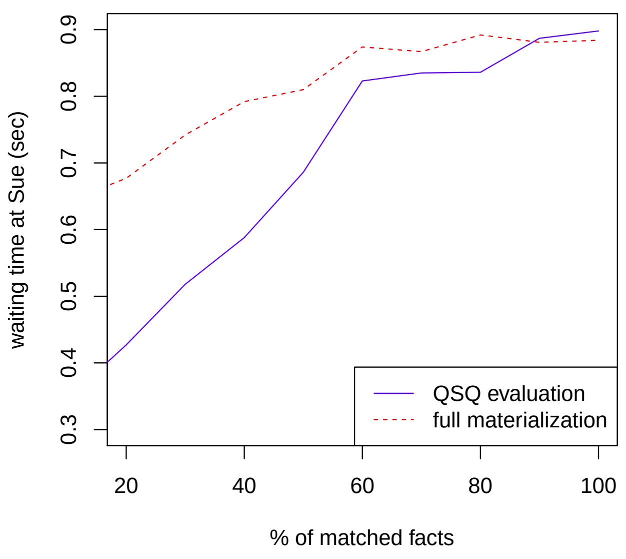

Figure 2: Distributed QSQ optimization

7 Related work

The *WebdamLog* language is motivated by previous work on the *WebdamExchange* system [5]. The system described there could automatically adapt to a variety of protocols and access methods found on the Web, notably for localizing data and for access control [10]. In developing toy applications with *WebdamExchange* [10], we realized the need for a logic that could be used (i) to *declaratively* specify applications and (ii) to exchange application logic between peers. This motivated the introduction of *WebdamLog* [4], a language based on rules that can run locally and be exchanged between peers.

Distributed data management has been studied since the earliest days of databases [20]. The fact that it is possible to access data from several data sources has been studied under various names, notably multi-databases or federated databases. The setting we consider is in the spirit of *peer-to-peer databases* with autonomous and heterogeneous data sources. Of course, standard query optimization techniques developed for distributed database systems are relevant here. We insist in particular on the techniques that are more relevant to our setting, which is based on datalog. One should mention that there have been a number of works on parallel or distributed evaluation of datalog, e.g., [2, 16].

The use of declarative languages, in particular datalog extensions, for distributed data management has already been advocated, e.g., in [3, 6]. There has recently been renewed interest in this approach [15]. Several systems have been developed based on the declarative paradigm [14, 18, 17], with performance comparable to that of systems based on imperative languages. Our implementation uses the *Bud* system [21]. The language Dedalus [9] has been proposed as a formal foundation for *Bud*. We prefer here to use the language *WebdamLog*, in particular because it features delegation.

Most classic optimization techniques for datalog are relevant to our work, in particular, semi-naive evaluation that is supported by *Bud*. We also considered the query-subquery optimization [22] as adapted to the distributed context in [2].

8 Conclusion

This paper presents an implementation of the *WebdamLog* language, introduced in [4]. The two main challenges for such an approach are (i) the difficulty of writing rules for non-technical users and (ii) the difficulty to offer good performance:

- With respect to (i), we present a user study that very promisingly shows that the participants (many of them not computer scientists) are able to understand and write simple rules.

- With respect to (ii), we benefit from previous datalog optimization techniques and efficient network communication by relying on the *Bud* system to support the basic functionality of distributed datalog. We show that the higher level features of *WebdamLog*, notably delegation, can be supported efficiently using logical rule rewriting.

All this demonstrates the feasibility of an approach based on *WebdamLog* to support exchanges of data and rules between rapidly evolving peers in a distributed and dynamic environment.

In the future, we are considering the following directions:

Access control One of the bases of *WebdamLog* is that a peer can locally install rules that are specified by another peer. Clearly, this is potentially very risky. Access control is therefore of paramount importance. We plan to work on access control, and in particular investigate the use of provenance for enforcing access control and for detecting access control violations.

Interface Our user study demonstrated that *WebdamLog* is appropriate for specifying distributed data management tasks. We are in the process of developing a user interface for the *WebdamLog* system. We also plan to conduct a follow-up user study (i) drawing from a larger pool of participants, (ii) including more participants without any CS training, and (iii) testing the usability of other aspects of the language, notably intentional vs. extensional predicates.

Application We intend to demonstrate the use of our system with complete applications, e.g., for social networks and personal data management.

Optimization We are currently developing a provenance-based approach for efficiently supporting changes in program state. Also, an optimization technique based on map-reduce and intense parallelism has been proposed for datalog [7]. It would be interesting to consider such an approach in our distributed setting.

References

[1] S. Abiteboul. Managing an XML warehouse in a P2P context. In *CAiSE*, pages 4–13, 2003. 1

[2] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *PODS*, pages 358–367, 2005. 8

[3] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008. 1, 8

[4] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *PODS*, 2011. 1, 2, 3, 5, 8

[5] S. Abiteboul, A. Galland, and N. Polyzotis. A model for web information management with access control. In *WebDB Workshop*, 2011. 8

[6] S. Abiteboul and N. Polyzotis. The data ring: Community content sharing. In *CIDR*, pages 154–163, 2007. 1, 8

[7] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8, 2011. 8

[8] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011. 5

[9] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, December 2009. 1, 3, 8

[10] E. Antoine, A. Galland, K. Lyngbaek, A. Marian, and N. Polyzotis. [Demo] Social Networking on top of the WebdamExchange System. In *ICDE*, 2011. 8

[11] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005. 1

[12] A. V. Gelder. Negation as failure using tight derivations for general logic programs. *J. Log. Program.*, 6(1&2):109–133, 1989. 1

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007. 6

[14] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *PADL*, pages 88–103, 2010. 8

[15] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010. 8

[16] M. A. W. Houtsma, P. M. G. Apers, and S. Ceri. Distributed transitive closure computations: The disconnection set approach. In *VLDB*, 1990. 8

[17] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, pages 97–108, 2006. 8

[18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, volume 39, pages 75–90, 2005. 8

[19] U. of Innsbruck. Iris - integrated rule inference system. `http://iris-reasoner.org/`. 4

[20] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011. 8

[21] B. O. O. M. project. Bloom programming language. `http://www.bloom-lang.net/`. 4, 8

[22] L. Vieille. Recursive Axioms in Deductive Databases: The Query/Sub-query Approach. In *EDS*, pages 253–267, 1986. 7, 8