A rule-based language for Web data management

Serge Abiteboul, Meghyn Bienvenu, Alban Galland, Émilien Antoine

A rule-based language for Web data management*

Serge Abiteboul
INRIA Saclay
& LSV-ENS Cachan
fname.lname@inria.fr

Meghyn Bienvenu
CNRS
& Université Paris-Sud
meghyn@lri.fr

Alban Galland
INRIA Saclay
& LSV-ENS Cachan
alban.galland@inria.fr

Émilien Antoine
INRIA Saclay
& Université Paris Sud
emilien.antoine@inria.fr

ABSTRACT

There is a new trend to use Datalog-style rule-based languages to specify modern distributed applications, notably on the Web. We introduce here such a language for a distributed data model where peers exchange messages (i.e. logical facts) as well as rules. The model is formally defined and its interest for distributed data management is illustrated through a variety of examples. A contribution of our work is a study of the impact on expressiveness of "delegations" (the installation of rules by a peer in some other peer) and explicit timestamps. We also validate the semantics of our model by showing that under certain natural conditions, our semantics converges to the same semantics as the centralized system with the same rules. Indeed, we show this is even true when updates are considered.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems, Distributed databases*

General Terms

Languages, Theory

Keywords

Datalog, Delegation, Distribution, Web

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

PODS'11, June 13–15, 2011, Athens, Greece.

1. INTRODUCTION

The management of modern distributed information, notably on the Web, is a challenging problem. Because of its complexity, there has recently been a trend towards using high-level Datalog-style rules to specify such applications. We introduce here a model for distributed computation where peers exchange messages (i.e. logical facts) as well as rules. The model provides a novel setting with a strong emphasis on dynamicity and interactions (in a Web 2.0 style). Because the model is powerful, it provides a clean basis for the specification of complex distributed applications. Because it is simple, it provides a formal framework for studying many facets of the problem such as distribution, concurrency, and expressivity in the context of distributed autonomous peers.

In recent years, there has been renewed interest in studying languages in the Datalog family for a wide range of applications ranging from program analysis, to security and privacy protocols, to natural language processing, or multiplayer games. For references see [17] and the forthcoming proceedings of the Datalog 2.0 workshop [15]. We are concerned here with using rule-based languages for the management of data in distributed settings, as in web applications [2, 7, 13, 4], networking [21, 20, 16] or distributed systems [22]. The arguments in favor of using Datalog-style specifications for complex distributed applications are the familiar ones, cf. [17].

A main contribution of the paper is a new model for distributed data management that combines in a formal setting deductive rules as in Datalog with negation, cf. [12] (to specify intensional data) and active rules as in Datalog$^{\neg\neg}$ [8] (for updates and communications). There have already been a number of proposals for combining active and deductive features in a rule-based language; see [19, 17] and our discussion of related work. However, there is yet to be a consensus on the most appropriate such language. We therefore believe that there is a need to continue investigating novel language features adapted to modern data management and to formally study the properties of the resulting new models.

The language we introduce, called Webdamlog, is tailored to facilitate the specification of data exchange between autonomous peers, which is essential to the applications we have in mind. Towards that goal, the novel feature we in-

troduce is *delegation*, that is, the possibility of installing a rule at another peer. In its simplest form, delegation is essentially a *remote materialized view*. In its general form, it allows peers to exchange rules, i.e., knowledge beyond simple facts, and thereby provides the means for a peer to delegate work to other peers, in Active XML style [3]. We show using examples that because of delegation, the model is particularly well suited for distributed applications, providing support for reactions to changes in evolving environments.

A key contribution is a study of the impact of delegation on expressivity. We show that view delegation (delegation in its simplest form, allowing only the specification of materialized views) strictly augments the power of the language. We also prove that full delegation further augments it. These results demonstrate the power of exchanging rules in addition to facts.

A message sent from peer p, received at peer q, that starts some task at q, introduces a kind of synchronization between the two peers. Thus, time implicitly plays an important role in the model. We show that when explicit time is allowed (each peer having its local time), view delegation no longer augments the expressive power of the language.

Because of their asynchronous nature, distributed applications in Webdamlog are nondeterministic in general. To validate our semantics for deductive rules, we study two kinds of systems that guarantee a form of convergence (even in presence of certain updates). These are positive systems (positive rules and persistence of extensional facts) and strongly-stratified systems (allowing a particular kind of stratified negation [12] for restricted deductive rules and fixed extensional facts). We also show that both types of systems essentially behave like the corresponding centralized systems.

The language Webdamlog is used in the implementation of a Web data management system that we briefly discuss. We also discuss some known optimization techniques that render this technology feasible.

Related work. This work is part of the ERC project *Webdam* on the foundations of Web data management [29]. The project is motivated by the conviction that the management of distributed data is still missing a unified formal foundation, and that this is hindering progress in Web data management. This opinion seems to be increasingly shared [17].

The present work is motivated by the renewed interest in distribution of Datalog. To our knowledge, the first attempts to distribute Datalog on different peers are [18] and [26]. The first distributes a positive Datalog program on different machines after a compilation phase. The second adapts classical transformations of positive programs based on semi-joins to minimize distribution cost.

Perhaps the most interesting usage of Datalog-style rules for distributed data management came recently from the Berkeley and U. Penn groups. They used distributed versions of Datalog to implement Web routers [24], DHT [23] and Map-Reduce [9] rather efficiently. By demonstrating what could be efficiently achieved with this approach, these works were essential motivations for our own. The most elaborate variant of distributed Datalog used in these works is presented in [21] and formally specified in [25]. In both papers, the semantics is operational and based on a distribution of the program before the execution. In view of issues with this model, a new model was recently introduced in [17], based on an explicit time constructor. We found the semantics of negation together with the use of time rather unnatural. In particular, time is used as an abstract logical notion to control execution steps and the future may have influence on the past. As a consequence, we found it difficult to understand what applications are doing as well as to prove results on their language. However, we have been influenced by this line of work.

We have also been influenced by previous work on Active XML [3], a model for distributed data intensive applications. Rules are used in Active XML [7], but they are different because the data are XML trees. Perhaps the work closest to that presented here is [1] that adapts query-subquery optimization [28] to a variant of positive distributed Datalog.

We are currently exploring a data model, called Webdam Exhange, for access control and distribution on the Web [5]. The model presented here is motivated to a large extent by the needs of Webdam Exchange applications.

Outline. The paper is organized as follows. We introduce the model in Section 2, first by means of examples and then formally. In the following section, we discuss some key features of the model and illustrate them with more examples. In Section 4, we compare the expressivity of different variants of our model. In Section 5, we discuss the convergence of Webdamlog systems and compare the semantics to the "centralized semantics", for the positive and strongly-stratified restrictions of the language. In Section 6, we mention an implementation and optimization techniques. The final section concludes with directions for future work.

2. THE MODEL

In this section, we first illustrate the model with examples, then formalize it. More examples and a discussion of key issues will be provided in the next section.

2.1 Informal presentation

Simple Webdamlog by example. We introduce with a first example the main concepts of the model: the notions of *fact* that captures both *local tuples* and *messages* between peers, *extensional* and *intensional* data, and *(Webdamlog) rule*.

Consider a particular peer, namely *myIphone*, with the relation *birthday* that gives the birthdates of friends and how to wish them a happy birthday (on which servers, with which messages). Examples of facts are:

birthday@myIphone("Alice", sendmail, inria.fr, 08/08)
birthday@myIphone("Bob", sms, BobIphone, 01/12)

Now the following [*Happy-Birthday*] rule is used to actually send birthday messages:

```
$message@$peer($name, "Happy birthday") :-
      today@myIphone($d),
      birthday@myIphone($name, $message, $peer, $d)
```

Observe that peer and message names are treated as data. The two previous *birthday*-facts represent pieces of local knowledge of *myIphone*. Now consider the fact :

sendmail@inria.fr(Alice, "Happy birthday")

This fact describes a message that is sent from *myIphone* to *inria.fr*.

As in deductive databases, the model distinguishes between extensional relations that are defined by a finite set of ground facts and intensional relations that are defined by rules. So for instance, the relation *birthday* on myIphone may be intensional and defined as follows:

```
intensional birthday@myIphone(string, relation, peer, date)
birthday@myIphone($n, $m, $p, $d) :-
    birthdates@myIphone($n, $d),
    contact@myIphone($n, $m, $p)
```

using extensional relations.

As usual, intensional knowledge is defined by rules such as the previous one, that we call *deductive* rules. Other rules such as the [*Happy-Birthday*] rule, that we call *active*, produce extensional facts. Such an extensional fact $m@p$ is received by the peer p (e.g. inria.fr and Bob's Iphone). During its next phase of local processing, this peer will consume these facts and produce new ones. By default, any processed fact disappears. Facts can be made persistent using persistence rules, illustrated next on the relation *birthdates@myIphone*:

```
birthdates@myIphone($n, $d) :-
    birthdates@myIphone($n, $d),
    ¬ del.birthdates@myIphone($n, $d)
```

The rules state that a fact persists unless there is explicitly a deletion message (e.g. *del.birthdates*).

Delegation by example. In the model, the semantics of the global system is defined based on local semantics and the exchange of messages and rules. Intuitively, a given peer chooses how to move to another state based on its local state (a set of personal facts and messages received from other peers) and its program. A move consists in (1) consuming the local facts, (2) deriving new local facts, which define the next state, (3) deriving nonlocal facts, i.e., messages sent to other peers, and (4) modifying their programs via "delegations".

The derivation of local facts and messages sent to other peers are both standard and were illustrated in the previous example. The notion of delegation is novel and is illustrated next. Consider the following rule, installed at peer p:

```
at p: m@q() :- m1@p($x), m2@p'($x)
```

where $m@q, m_1@p$ and $m_2@p'$ are all extensional. Its semantics is as follows. Suppose that we have a value a_1 such that $m_1@p(a_1)$ holds, then the effect of this rule is to install at p' the following rule:

```
at p': m@q() :- m2@p'(a1)
```

The action of installing a rule at some other peer is called *delegation*. When p' runs, if $m_2@p'(a_1)$ holds, it will send the message $m@q()$ to q.

Now suppose instead that $m@q$ is intensional. When p' runs, if $m_2@p'(a_1)$ holds, the effect of this rule is to install at q the following rule:

```
at q: m@q() :-
```

The intuition for the delegation from p to p' is that there is some knowledge from p' that is needed in order to realize the task specified by this particular rule. So, to perform that task, p delegates the remainder of the rule to p'. The delegation from p' to q is somewhat different. Peer p' knows that $m@q$ (an intensional fact) holds until some change occurs. As q may need this fact for his own computation, p' will pass this information to q in the form of a rule (since as a fact, it would be consumed).

We next formalize the model illustrated by the previous example.

2.2 Formal model

Alphabets. We assume the existence of two infinite disjoint alphabets of sorted *constants*: *peer* and *relation*. We also consider the alphabet of *data* that includes in addition to *peer* and *relation*, infinitely many other constants of different sorts (notably, *integer*, *string*, *bitstream*, etc.). It is because *data* includes *peer* and *relation* that we may write facts such as those in the *birthday* relation. Similarly we have corresponding alphabets of sorted *variables*. An identifier starting by the symbol \$ implicitly denotes a variable. A *term* is a variable or a constant.

A *schema* is an expression $(\Pi, \mathcal{E}, \mathcal{I}, \sigma)$ where Π is a (possibly infinite) set of peer IDs; $\mathcal{E}$ and $\mathcal{I}$ are disjoint sets, respectively, of *extensional* and *intensional* names of the form $m@p$ for some relation name m and some peer p; and the typing function σ defines for each $m@p$ in $\mathcal{E} \cup \mathcal{I}$ the arity and sorts of its components. Note because $\mathcal{I} \cap \mathcal{E} = \emptyset$, no m is both intensional and extensional in the same p. Considering Π to be infinite reflects the assumption that the set of peers is dynamic and of unbounded size just like data.

Facts and rules. Given a relation $m@p$, a (ground) (p-)*fact* is an expression $m@p(\overline{u})$ where $\overline{u}$ is a vector of data elements of the proper type, i.e., correct arity and correct sort for each component. For a set K of facts and a peer p, $K[p]$ is the set of p-facts in K. The notion of fact is central to the model. It will be the basis for both stored knowledge and communication. For instance, in the peer p, if we derive the extensional fact $r@p(1,2)$, this is a fact p knows. On the other hand, if we derive the extensional fact $s@q(2,3)$, this is a message that p sends to q.

A *(Webdamlog) rule* is an expression of the form

$$M_{n+1}@Q_{n+1}(\overline{U}_{n+1}) \text{ :- } (\neg)M_1@Q_1(\overline{U}_1)...(\neg)M_n@Q_n(\overline{U}_n)$$

where each M_i is a relation term, each Q_i is a peer term and each $\overline{U}_i$ is a vector of data terms. We also allow in the body of the rules, atoms of the form $X = Y$ or $X \neq Y$ where X, Y are terms.

We require a rule to be *safe*, i.e.,

1. For each i, if Q_i is a peer variable, it must be previously bound, i.e. it must appear in $\overline{U}_j$ for some positive literal $M_j@Q_j(\overline{U}_j), j < i$.
2. Each variable occuring in a literal $\neg M_i@Q_i(\overline{U}_i)$ must be previously bound to a positive literal.
3. Each variable in the head must be positively bound in the body.

REMARK 1. *Observe that we treat differently peer and relation names. By (1), a peer variable has to be previously positively bound. We insist on (1) so that we control explicitly to whom a peer sends a message or delegates a rule.*

Note also that because of (1), the ordering of literals is relevant. One could define a variation of the language, namely

peer-unguarded Webdamlog *by not imposing Constraint (1) and considering all orderings of the body literals (with the negative ones seen implicitly after all the others). When deriving new facts, we simply consider first the positive literals and never consider a literal if its peer is not instantiated. This variant would not differ much from the language we study here.*

We say that a rule is *deductive* if the head relation is intensional. Otherwise, it is *active*. Rules live in peers. We say that a rule in a peer p is *local* if all Q_i in all body relations are from p. It is *fully local* if the head relation is also from p. We will see that the following four classes of rules play different roles:

Local deduction Fully local deductive rules are used to derive intensional facts *locally*.

Update Local active rules are used for sending messages, i.e., facts, that modify the databases of the peers that receive them.

View delegation The local but not fully local deductive rules provide some form of view materialization. For instance, this rule results in providing at q a view of some data from p:

$$at\ p:\ r@q(\overline{U}) \text{ :- } (\neg)r_1@p(\overline{U_1}), ...(\neg)r_n@p(\overline{U_n})$$

General delegation The remaining rules allow a peer to install arbitrary rules at other peers.

Peer and relation variables provide considerable flexibility for designing applications. However, observe that because of them, it may be unclear whether a rule is (fully) local or not, deductive or active. Using atoms such as $Q = p$, $Q \neq p$ for some constant peers and similarly for relations, one could remove the ambiguity and distinguish the nature of the rule. We omit the formal details. Note that in a real system, one can wait until a rule is (partially) instantiated at runtime to find what its nature is, and decide what should be done with it.

The semantics of Webdamlog is based on autonomous local computations of the peers. We consider this first, then look at the global semantics of Webdamlog systems.

Local computation. A local computation happens at a particular peer. Based on its set of facts and set of rules, the peer does the following: (1) some local deduction of intensional facts, (2) the derivation of extensional facts that either define its next state or are sent as messages, and (3) the delegation of rules to other peers.

(Local deduction) For local deduction, we want to rely on the semantics of standard Datalog languages. However, because of possible relation variables, Webdamlog rules are not strictly speaking proper Datalog$^\neg$ rules, since the relation names of atoms may include variables. So, to specify local deduction, we proceed as follows. We start by *grounding* the peer and relation variables appearing in the rules. More precisely, for each rule

$$M_{n+1}@Q_{n+1}(\overline{U}_{n+1}) \text{ :- } (\neg)M_1@Q_1(\overline{U}_1)...(\neg)M_n@Q_1(\overline{U}_n)$$

of peer p, we consider the set of rules obtained by instantiating relation variables M_i with relation constants and peer variables Q_i with peer constants. To ensure finiteness, we only use constants from the active domain of the peer, that is, that appear in some fact or rule in the peer state. We can now treat pairs $m@p$ of relation and peer constants as normal relation symbols in Datalog. Since for local deduction, we are only interested in fully local deductive rules, we will remove rules with a relation $m@q$ for $q \neq p$ or an extensional relation in the head. We must also remove rules that violate the arity or sort constraints of σ. The remaining rules are all fully local deductive rules which belong to standard Datalog.

Now, given a set I of facts and a set P_d of fully local deductive rules (obtained as in the preceding paragraph), we denote by $P_d^*(I)$ the set of facts inferred from I using P_d with a standard Datalog semantics. For instance, in absence of negation, the semantics is, as in classical Datalog, the least model containing I and satisfying P_d. When considering negation, one can use any standard semantics of Datalog with negation, say well-founded [27] or stable [14]. For results in Section 5.2, we will use a variant of stratified negation semantics [12].

(Updates) Given a set K of facts and a set P_a of local active rules, the set $P_a(K)$ of *active consequences* is the set of *extensional* facts $v(A)$ such that for some rule A :- Θ of P_a and some valuation v, $v(\Theta)$ holds in K, and $v(A), v(\Theta)$ obey the typing and sort constraints of σ. This is the set of *immediate consequences*. Note that it does not necessarily contain all facts in K.

Observe that for deductive rules, we typically use a fixpoint (based on the particular semantics that is used), whereas for active rules, we use the immediate consequence operator that is explicitly procedural.

(Delegation) Given a set K of facts and a set P of (active and deductive) rules in some peer p, the *delegation* $\gamma_{pq}(P, K)$ of peer p to $q \neq p$ is defined as follows.

If for some deductive rule $M@Q(\overline{U})$:- Θ in P, there exists a valuation v such that $v\Theta$ holds in K, $v(Q) = q$, and the typing constraints in σ are respected, then

$$vM@vQ(v\overline{U}) \text{ :-}$$

is in $\gamma_{pq}(P, K)$.

If for some active or deductive rule

$$A \text{ :- } \Theta_0, (\neg)M@Q(\overline{U}), \Theta_1$$

in P (where Θ_0, Θ_1 are sequences of possibly negated atoms), there exists a valuation v satisfying σ such that $v\Theta_0$ contains only p-facts, $v\Theta_0$ holds in K, and $vQ = q(\neq p)$, then

$$vA \text{ :- } (\neg)M@vQ(v\overline{U}), v\Theta_1$$

is in $\gamma_{pq}(P, K)$.

Nothing else appears in $\gamma_{pq}(P, K)$.

Observe that we do not produce facts that are improperly typed. In practice, a peer p may not have complete knowledge of the types of some peer q's relations. Then p may "derive" an improperly typed fact. This fact will be sent and rejected by q. From a formal viewpoint, it is simply assumed that the fact has not even been produced. Similarly, a peer may delegate an improperly typed rule, but that rule will never produce any facts, and so can safely be ignored.

We are now ready to specify the semantics of Webdamlog systems.

States and runs. A *(Webdamlog) state* of the schema $(\Pi, \mathcal{E}, \mathcal{I}, \sigma)$ is a triple $(I, \Gamma, \tilde{\Gamma})$ where for each $p \in \Pi$, $I(p)$ is a finite set of extensional p-facts at p, $\Gamma(p)$ is the finite set of rules at p, and $\tilde{\Gamma}(p,q)$ $(p \neq q)$ is the set of rules that p delegated to q. For $p \in \Pi$, the (p-)*move* from $(I, \Gamma, \tilde{\Gamma})$ to $(I', \Gamma', \tilde{\Gamma}')$ (corresponding to the *firing* of peer p) is defined as follows. Let P_p be $\Gamma(p) \cup (\cup_q \tilde{\Gamma}(q,p))$, P_{pd} be the set of fully local deductive rules in P_p and P_{pa} the set of local active rules in it. Then the next state is defined as follows:

- (Local deduction) Let $K = P^*_{pd}(I(p))$.
- (Updates) $I'(p) = P_{pa}(K)[p]$; and
  (external activation) $I'(q) = I(q) \cup P_{pa}(K)[q]$ for each $q \neq p$.
- (Delegations) $\tilde{\Gamma}'(p,q) = \gamma_{pq}(P_p, K)$ for each q $\neq$ p; and $\tilde{\Gamma}'(p',q') = \tilde{\Gamma}(p',q')$ otherwise.

A (Webdamlog) *system* is a state $(I, \Gamma, \tilde{\Gamma})$ where $\tilde{\Gamma}(p,q) = \emptyset$. We will speak of the system (I, Γ) (since $\tilde{\Gamma}$ is empty). A sequence of moves is *fair* if each peer p is invoked infinitely many times. A *run* of a system (I, Γ) is a fair sequence of moves starting from (I, Γ).

Observe that $I(p)$ is finite for each peer and that it remains so during a run, even if the number of peers is infinite. Note also that deletions are implicit: a fact is deleted if it is not derived for the next state. We recall that facts can be made persistent using persistence rules of the form

$$r@p(\overline{U}) \text{ :- } r@p(\overline{U}), \neg del.r@p(\overline{U})$$

In the following, such a rule for relation $r@p$ will be denoted *persistent r@p*.

REMARK 2. *It is important to observe a difference between the semantics of facts and rules. Observe that, if we visit twice peer p in a row, the fact-messages that p sends to q accumulate at q. On the other hand, the new set of delegations replaces the previous such set. Moreover, when we visit q, the messages of q are consumed whereas the delegations stay until they are replaced. These subtle differences are important to capture different facets of distributed computing, e.g., for capturing materialized views or for providing the expected semantics to extensional / intensional data.*

3. DISCUSSION

In this section, we present examples that illustrate the interest of our model for distributed data management, and make key observations about different aspects of the model.

We first consider two serious criticisms that could be adressed to the model, namely too much synchronization and too little local control. We show how both issues can be resolved.

Too much synchronization. Observe that moves capture some form of asynchronicity and parallelism. The peer that fires is randomly chosen and does (atomically) some processing. However, there is still some undesired form of synchronization. When we process peer p, messages from p to some peer q are instantaneously available in q. This is impossible to guarantee in practice. In a standard manner, when a more precise modeling is desired, one can introduce a peer acting as the network between p and q. Instead of going instantaneously from p to q, the message goes instantaneously from p to $network_{pq}$, waits there until $network_{pq}$ is fired, then goes instantaneously to q, and similarly for delegations. This captures more realistically what happens in practice, and does not require changing the model.

Too little local control. In the model we have defined, nothing prevents a peer p from modifying another peer q's relations or accessing q's data using delegation. In realistic settings, one would want a peer to be able to hold private information, which cannot be modified or accessed by another peer without its permission. This can be easily accomplished by extending the model with *local relations*. These relations can only appear in p's own facts and rules (i.e. $I(p)$ and $\Gamma(p)$), but not in any rules delegated to p (in practice, this means p would simply ignore any delegations using one of its private relations).

To illustrate, suppose that we want to control the access to a relation $r@p$ of peer p. We create for this purpose two local relations $read@p(\$r, \$q)$ and $write@p(\$r, \$q)$ that store who can read/write in p's relations. Note that the *read* and *write* relations are local, i.e., only p can specify the access rights in p. Relations $r@p$ and $del.r@p$ must also be local so that p control access to them. To obtain relation $r@p$, a peer q sends a message $get@p(r,q)$. The following rule controls whether q will receive the data it requested:

```
at p: send@$q($r,$x) :- get@p($r,$q), read@p($r,$q),
                        $r@p($x)
```

Insertions in $r@p$ (or deletions using $del.r@p$) are treated similarly. Access control is at the center of the work around WebdamExchange [5] that motivated the work presented here.

We next consider two subtleties of delegation.

Delegation and complexity. Consider the rule:

```
at p: m@q() :- m1@p($q,$x), m2@$q($x)
```

If there are 1000 distinct tuples $(p_i, 0)$ such that $m_1@p(p_i, 0)$ holds, then we have to install rules in 1000 distinct peers. So delegation is inherently transforming data complexity into program complexity.

Peer life and delegation. It is very simple in the model to consider that peers are born, die or hibernate. We simply have to insist that p can be fired (p-move) only if p is alive and not hibernating. We can assume that messages and delegations to dead peers are simply lost and that for hibernating ones, they are buffered somewhere in the network. A subtlety is that (with this variant of the model), if a peer dies without cleanly terminating, delegations from this peer are still valid. In practice, the system may realize that a particular peer is no longer present and terminate its delegations.

We conclude this section with three examples that illustrate different aspects of the language, communications, persistence services, and rule updates.

Multicasting. We can simulate channels, i.e., m-n communications with the following rules:

```
at q: persistent channelsubscribe@q
      channel@$p($m,q,$s) :- channelsubscribe@q($p,$m),
          $m@q($s)
```

The rules at peer q allows him to support channels. A peer p can subscribe to receiving all the messages from the channel m hosted by q by sending $channelsubscribe@q(p, m)$ to q. Then, whenever someone sends a message $m@q(s)$, p will receive $channel@p(m, q, s)$.

Database server replication. The following rule allows a database server to replicate relations from many peers:

```
intensional export@db(relation,peer)
at db: persistent tobeexported@db
       export@db($r,$p,$x) :- tobeexported@db($r, $p),
           $r@$p($x)
```

If a peer p wants his relation $r@p$ to be stored at db, then p simply needs to send q the message $tobeexported@db(r, p)$. Now, $export@db(r, p, \$x)$ is a copy of $r@p(\$x)$.

Rule updates and rule deployment. Observe that (to simplify) we assumed that the set of rules in a run is fixed, i.e., $\Gamma(p)$ is fixed for each p. It is straightforward to extend the model to support addition or deletion of rules. Furthermore, one might want to be able to control whether a particular rule is deployed on a particular peer. To illustrate this point, consider the two rules:

```
at p: persistent server@p
      f@$p($u) :- server@p($p), f1@$p($u1),...,fn@$p($un)
```

Sending the message $server@p(q)$ results in installing

```
at q: f@q($u) :- f1@q($u1),...,fn@q($un)
```

Note that if we send the message $del.server@p(q)$, the rule is removed.

4. EXPRESSIVITY

In this section, we study the expressive power of Webdamlog and of different languages that are obtained by allowing or restricting delegations. We also consider the expressive power of timestamps. More precisely, we consider the following languages for rules:

- WL (Webdamlog): the general language.
- VWL (views WL): the language obtained by restricting delegations to only view delegations.
- SWL (simple WL): the language obtained by disallowing all kinds of delegations.

At the core of view delegation, we find the maintenance of materialized views. To maintain views, we will see that timestamps turn out to be useful. More precisely, for time, we assume that each peer has a local predicate called *time* (with $time(t)$ specifying that the current move started at local time t.) The predicate $<$ is used to compare timestamps. Note that each peer has its separate clock, so the comparison of timestamps of distinct peers is meaningless. To prevent time from being a source of nondeterminism, for t_1, t_2 two times at different peers, we have: $t_1 \not< t_2$ and $t_2 \not< t_1$. The languages obtained by extending the previous languages with timestamps are denoted as follows: WL^t, VWL^t, SWL^t.

Traces and simulations. To formally compare the expressivity of the above languages, we need to introduce the auxiliary notions of trace and simulation.

Let $r = (I_1, \Gamma_1, \widetilde{\Gamma}_1), ...(I_n, \Gamma_n, \widetilde{\Gamma}_n), ...$ be a run. Let M be a set of predicates and I a set of facts. Then $\Pi_M(I)$ is the set of facts in I with predicates in M. The *M-trace* of the run r for a set M of predicates is the subsequence of $\pi_M(I_{i_1}), ..., \pi_M(I_{i_n})...$ obtained by starting from $\pi_M(I_1), ..., \pi_M(I_n)...$ and removing all repetitions, i.e., deleting the $(k+1)$th element of the sequence if it is identical to the kth, until the sequence does not contain two identical consecutive elements. Given an initial state S and a set of predicates M, we denote by $M\text{-}trace(S)$ the set of M-traces of runs from S. In some sense, it is what can be observed from S when only facts over M are visible.

Le α be a set of peers. An initial state $S = (I, \Gamma)$ can be *α-simulated* by an initial state $S' = (I, \Gamma')$ if $\Gamma(p) = \Gamma'(p)$ for all $p \in \alpha$ and S and S' have the same M-traces, where M is the set of relations of S. In other words, from the point of view of what is visible from S, S' behaves exactly like S. The set of peers α is meant to capture the part of the system (one or more peers) that we want to keep strictly identical.

Now, we say that a language L can be *simulated* by a language L', denoted $L \prec L'$, if there exists a translation τ from programs in L to programs in L' such that for each initial state (I, Γ) (with programs in L) and for each α, $(I, \overline{\tau}(\Gamma))$ α-simulates (I, Γ) where $\overline{\tau}$ is defined by: for each peer p,

- if $p \in \alpha$, $\overline{\tau}(\Gamma(p)) = \Gamma(p)$.
- otherwise, $\overline{\tau}(\Gamma(p)) = \tau(\Gamma(p))$.

Clearly, in the previous definition, the peers in α are not part of the simulation, they behave exactly as originally. In some sense, they should not even be aware that something changed.

Expressivity results. The expressive power of the different languages are compared in Figure 1. The containments are strict except for that of VWL^t inside WL^t where the issue remains open.

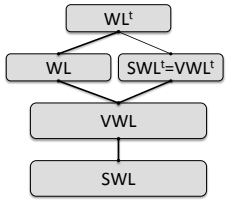

Figure 1: Expressive power of the rule languages (the inclusion is strict when the arc is in bold)

Our first result states that view delegation cannot be simulated by simple rules.

THEOREM 1 (NO VIEWS IN SWL). *VWL $\not\prec$ SWL.*

PROOF. Intuitively, the difficulty is that the system may visit an arbitrary number of times the same peer p before visiting another peer q. Then q sees all the messages from

p at the same time and ignores in which order they were received.

Formally, consider a VWL system (I, Γ) consisting of three peers p_α, p, q. There are two facts that hold in the initial state: $true@p_\alpha()$, $true@p()$.

The set of active rules $\Gamma(p_\alpha)$ maintain the peer p_α in a permanent flip-flop between two modes:

$$\begin{aligned} at\ p_\alpha : r@p() &\text{ :- } true@p_\alpha() \\ false@p_\alpha() &\text{ :- } true@p_\alpha() \\ del.r@p() &\text{ :- } false@p_\alpha() \\ true@p_\alpha() &\text{ :- } false@p_\alpha() \end{aligned}$$

Note that p_α keeps inserting then deleting the same proposition in p, namely $r@p()$. Peer p uses the following four rules:

$$\begin{aligned} at\ p : r@p() &\text{ :- } r@p(), \neg del.r@p() \\ true@p() &\text{ :- } false@p() \\ false@p() &\text{ :- } true@p() \\ s@q() &\text{ :- } r@p() \end{aligned}$$

The first active rule maintains relation $r@p$. The next two active rules maintain p in a flip-flop between two modes. The last rule is a view delegation rule. It is because of this latter rule that the system is in VWL but not in SWL.

Finally peer q has one active rule:

$$at\ q : true@q() \text{ :- } s@q()$$

Suppose for a contradiction that there is a p_α-simulation of this system in SWL, via some program translation function τ. As the set of peers is finite (namely 3), the initial state $(I, \tau(\Gamma))$ is finite. Thus, it includes a finite set of relation names and constants. This means that there is a finite number of distinct messages that can be sent during a run of this system. Now let r_1 be any run of $(I, \tau(\Gamma))$ such that the initial segment of activated peers is as follows: p_α, then p, then p_α, then p, etc., n times (for n to be fixed later in the proof), and then q. Let $I, I_1, I_2, ..., I_{2n-1}, I_{2n}, I'$ be the trace of r_1. Because of the two flip-flops, the trace has this size and it is clear from it which peer has been activated at each step.

Consider a second run r_2 which is defined like r_1 except that this time we visit p_α and p, $n+1$ times, then q. Let $I, I_2, I_3..., I_{2n-1}, I_{2n}, I_{2n+1}, I_{2n+2}, I''$ be the trace of r_2.

Observe that while p and p_α are being activated, q is simply accumulating messages. Recall that the set of messages that q may accumulate is finite. Thus we can choose n large enough so that $I_{2n+2}(q) = I_{2n}(q)$. Suppose that $I'(q)$ contains $true@q$. Then because the set of messages at q is the same in the second run, $I''(q)$ also contains $true@q$, a contradiction because the last iteration in p_α, p must have removed $r@p$. A similar contradiction occurs if $true@q$ is not produced. Thus such a simulation does not exist. □

Next we separate VWL and WL.

THEOREM 2 (NO GENERAL DELEGATIONS IN VWL). *WL $\not\prec$ VWL.*

PROOF. (sketch) Intuitively, peer q will use a general delegation to ask peer p to do something that is beyond the capability of the rules in p. This is not trivial because p may perform very complex operations with arbitrarily many complex rules. However, it turns out that there is a limit to what p can do. To prove it, we use the fact that with formulas using a bounded number k of variables, one cannot check whether a graph has a clique of size $k+1$ (when an ordering of the nodes is not available).

Formally, consider a WL system (I, Γ) that consists of three peers p_α, p, q. Intuitively, peer p_α sends a sequence of updates to a graph that is originally empty and is stored at p. To do that, p_α has a persistent relation that stores a sequence of updates. More precisely, p_α has a set of tuples of the form: $upd@p_\alpha(i, o, a, b)$ where i in [0,m] for some m and there is a single tuple for each i, o in { ins, del }, and a, b are data elements in a very large fixed set Σ (the identifiers of the graph g.) Peer p_α also has a persistent relation *next* containing the tuples: $[0,1], ...[m-1, m]$. Finally, p_α has the fact $now@p_\alpha(0)$ in its initial state. The program of p_α consists of the following active rules:

$$\begin{aligned} at\ p_\alpha : g@p(\$x, \$y) &\text{ :- } now@p_\alpha(\$i), upd@p_\alpha(\$i, ins, \$x, \$y) \\ del.g@p(\$x, \$y) &\text{ :- } now@p_\alpha(\$i), upd@p_\alpha(\$i, del, \$x, \$y) \\ now@p_\alpha(\$j) &\text{ :- } now@p_\alpha(\$i), next(\$i, \$j) \end{aligned}$$

Now p has the following active rule for maintaining the graph g :

$$at\ p : g@p(\$x, \$y) \text{ :- } g@p(\$x, \$y), \neg del.g@p(\$x, \$y)$$

Finally, peer q has a rule delegation to p:

$$at\ q : clique@q() \text{ :- } \wedge_{1 \le i,j \le n} g@p(\$x_i, \$x_j), \$x_i \neq \$x_j$$

which essentially requests p to send a message if there exists an n-clique in $g@p$. Peer q also has a flip-flop rule:

$$\begin{aligned} at\ q : true@q() &\text{ :- } false@q() \\ false@q() &\text{ :- } true@q() \end{aligned}$$

Originally $true@q()$ holds.

Suppose for a contradiction that there is a p_α-simulation of this system in *VWL*. Consider the run of (I, Γ) beginning with a very long sequence $q(p_\alpha)^* p(p_\alpha)^* ... p$ where each time p is called, the graph oscillates between "there is a clique" and "there isn't". Note that the first time q is called, it installs the delegation.

Let k be the number of variables and constants that appear in a rule in $\tau(\Gamma(p))$. As the rules in p have less than k symbols, they can only evaluate formulas in FO^k. Choose $n > k$, so that formulas in FO^k cannot check for the presence of an n-clique in a graph. Choose also the set of graph identifiers Σ large enough. (Recall that the translation for the rules of p is independent from the program of q and p_α.) So, it is not possible for p to evaluate whether there is a clique. So q has to be called before each *clique* message to check the existence of a clique. Note that it is possible to do so: p pretends it has not been called and waits until q is called; then q sends a secret message to p to tell p whether there is a clique.

This is "almost" a simulation except that q has a bounded memory that depends essentially on Σ. Now consider a very long sequence of the WL system that never calls q. If the sequence is long enough, its simulation in VWL will visit twice the same state. Then by pumping, one can construct an infinite run of the VWL simulating system such that the flip-flop of q is never activated. This corresponds to a simulation of an unfair run of the WL system, a contradiction. Thus there can be no VWL simulation of the above WL system. □

We now consider timestamps. The next result compares the expressive power of WL and WL^t.

THEOREM 3 (TIMESTAMPS). *For a finite number of peers,*

1. *WL is in* PSPACE*;*
2. *SWL^t over a single peer can simulate any arbitrary Turing machine;*
3. *Thus, SWL^t $\not\approx$ WL and (a fortiori) WL^t $\not\approx$ WL.*

PROOF. (sketch) For (1.), consider a fixed schema over a finite number of peers. Let (I, Γ) be an initial instance of size $n = |I| + |\Gamma|$. Let $(I_i, \Gamma, \widetilde{\Gamma}_i)$ be an instance that is reached during the computation. Because the schema is fixed, the number of facts that can be derived is bounded by a polynomial in n, and each fact is also of bounded size. So, $|I_i|$ can be bounded by a polynomial in n. Similarly, the size of $\widetilde{\Gamma}_i$ can be bounded by a polynomial in n, since a rule that is delegated is essentially determined by an instantiation of an original rule and a position in it. Thus we can represent $(I_i, \Gamma, \widetilde{\Gamma}_i)$ in polynomial space in n. Hence, WL is in PSPACE.

Now consider (2.). Let M be a Turing Machine. We can assume without loss of generality that it is deterministic and that it has a tape that is infinite only in one direction. The SWL^t system that simulates it is as follows. Its initial instance encodes the initial state of M. More precisely, it has a relation *input*, with initial value

$$\{\ input(0,1,a_1),\ input(1,2,a_2),\ \ldots\ input(n{-}1,n,a_n)\ \}$$

where $a_1 a_2 ... a_n$ is the input of M. It also has a relation *tape* that is originally empty.

First, the SWL^t system copies the input on its tape using the timestamps $t_0, t_1, t_2 ...$ to identify tape cells. More precisely, it constructs,

$$\{tape(t_0,t_1,a_1,s_0), tape(t_1,t_2,a_2,\bot), ..., tape(t_{n-1},t_n,a_n,\bot)\}$$

where s_0 is the start state of M. Using rules from SWL^t, it is straightforward to simulate moves of M. The only subtlety is that at each step of the iteration, the tape is augmented so that there is no risk of reaching its limit. The fact that the cells are denoted with timestamps guarantees that no two cells will have the same ID.

Now, given the encoding of a word w, one can simulate the computation of TM on w. Thus (2), so (3). □

Note that the converse of (1) holds: any PSPACE query over an ordered database can be computed in SWL (hence WL) with a single peer. This can be shown by proving how to simulate in SWL with a single peer, the language Datalog$^{\neg\neg}$ that can express all PSPACE queries on ordered databases [8].

Next we see how to use timestamps to simulate view maintenance.

THEOREM 4 (VIEWS WITH TIMESTAMPS). *VWL$^t \approx$ SWL^t.*

PROOF. (sketch) We illustrate with an example the simulation of view delegation by a program with timestamps.

Consider a VWL system with an extensional relation $s@q$ and the deductive rule at p: $r@p(\overline{U})$:- $s@q(\overline{U})$ that specifies that $r@p$ is a view of $s@q$. The simulation of the view delegation in SWL^t is as follows.

at q : persistent past@q
  $aux@p(\overline{U}, \$t)$:- $s@q(\overline{U})$, $time@q(\$t)$
  $past@q(\$t)$:- $time@q(\$t)$
  $obsolete@p(\$t)$:- $past@q(\$t)$

at p : intensional r@p
  persistent aux@p, obsolete@p
  $r@p(\overline{U})$:- $aux@p(\overline{U}, \$t)$, $\neg$ $obsolete@p(\$t)$

Then the value of $r@p$ is that of $s@q$ when q was last visited, i.e., $r@p$ is a copy of $s@q$ at the last visit of q.

The above simulation is straightforwardly generalized to arbitary VWL systems, from which we obtain the desired VWL$^t \approx$ SWL^t. □

It is still open whether WL$^t \not\approx$ VWL^t.

5. CONVERGENCE OF WEBDAMLOG

Systems that converge to a unique state independently of the order of computation, i.e., some form of Church-Rosser property, are of particular interest. In this section, we consider two kinds of such systems: the positive and the strongly-stratified Webdamlog systems. Indeed, we show that such systems continue to converge even in presence of insertions of facts or rules. Finally, we show that for these two classes of systems, the distributed semantics can be seen as mimicking the centralized semantics.

5.1 Positive Webdamlog

Clearly, negation may explain why a system does not converge. However, the following example shows that even in absence of negation, convergence is not guaranteed because the order of arrival of messages matters:

EXAMPLE 1. *Consider the rules:*

at q: extensional r1@q, r2@q, r@q
  persistent r@q
  r@q() :- r1@q(), r2@q()
at q1: r1@q() :-
at q2: r2@q() :-

If we process the peers according to the order $q1, q, q2, q, q1, \ldots$, then $r@q$ is never derived. If we consider instead the order $q1, q2, q, q1, q2, q, \ldots$, then $r@q$ is derived and remains forever. The absence of convergence here is in fact a desired feature of the model: the extensional relations model events, so their arrival times matter.

On the other hand, note that, as we will see, if in the example $r1@q$ and $r2@q$ were intensional, the system would converge.

We now introduce the restricted systems we study in this section. A Webdamlog state or system is *positive* if the following holds:

1. Each of its rules is positive (no negation); and
2. Each extensional relation $m@p$ is made persistent with a rule of the form $m@p(\overline{U})$:- $m@p(\overline{U})$.

We will see that because of these restrictions, the states in runs of positive systems are monotonically increasing. For positive systems with a finite number of peers, there are only finitely many possible states, so monotonicity implies that runs converge after a finite number of steps. We will also show convergence for positive systems with infinitely many peers, except that in this case, we may converge only in

the limit. This motivates the following somewhat complex definition of convergence.

A run $S_0, S_1, S_2, \ldots$ *converges* to a possibly infinite state $S^* = (I^*, \Gamma^*, \widetilde{\Gamma}^*)$ if for each finite $S' \subseteq S^*$, there exists $k_{S'}$ such that for all $k > k_{S'}$, $S' \subseteq S_k$ and if for each finite $S' \not\subseteq S^*$, there is $k_{S'}$ such as for all $k > k_{S'}$, $S' \not\subseteq S_k$. We say a system S *converges* if all its runs converge to the same state.

The following theorem states the convergence of (possibly infinite) positive systems.

THEOREM 5 (CONVERGENCE). *All positive Webdamlog systems converge.*

The previous theorem is still true if one allows the peers to insert facts and rules. One can show that the system will reach a stable state that does not depend on the points of insertion.

THEOREM 6 (UPDATES). *Given two positive Webdamlog systems (I,Γ) and (I′,Γ′), for any run of the system (I,Γ), if for a given step, I′ is added to the current set of facts and Γ′ to the current set of rules, then the modified run converges to the convergence state of (I ∪ I′,Γ ∪ Γ′).*

The previous theorem is straightforwardly extended to a series of updates. However, as illustrated by the following example, a more liberal definition of updates which also allows deletion of facts or rules in a system would compromise convergence.

EXAMPLE 2. *Consider the system defined as follows:*

at p: extensional@p, intensional r@p
r@q() :- r@p()
r@p() :- s@p()
s@p() :- s@p()
s@p().
at q: intensional r@q
r@p() :- r@q()

This system converges to a state where $I^*(p) = \{s@p()\}$, $\widetilde{\Gamma}^*(p,q) = \{r@q()\text{:-}\}$, $\widetilde{\Gamma}^*(q,p) = \{r@p()\text{:-}\}$ *Then removing the fact s@p() or the rule r@p():- s@p() after the convergence will not change* $\widetilde{\Gamma}$ *whereas* $\widetilde{\Gamma}$ *would be empty were the fact or the rule removed before beginning a run.*

The previous example illustrates the difficulty of managing non-monotony. If we remove a fact or a rule, we need to remove as well all facts or rules that were deduced using this fact. This could be achieved using view maintenance techniques. We leave this to future work.

To further ground our semantics, we show that for positive systems, our semantics correspond to the standard centralized Datalog semantics.

Centralized semantics. In the positive case, we can compare with a "centralized" semantics, in which all facts and rules are combined into a single Datalog program. Such a comparison would not make sense in the general case since our semantics too closely depends on the order in which peers fire.

We associate to a positive Webdamlog state (I, Γ) the set $\cup_p(I(p) \cup \Gamma(p))$ composed of the facts and rules of all peers. We can transform this set of facts and rules into a standard Datalog program by first instantiating the variable relations in the rules (as was done for local computation) and then removing those rules which violate the typing constraints in σ. We denote by $c(I, \Gamma)$ the Datalog program thus obtained.

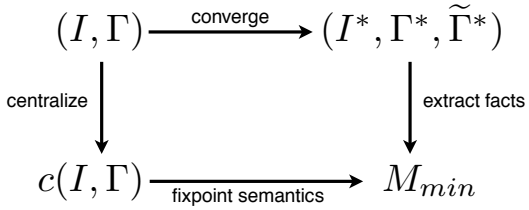

Figure 2: Link with centralized semantics

The following theorem (illustrated by Figure 2) demonstrates the equivalence, for the class of positive systems, of our distributed semantics and the traditional fixpoint semantics of Datalog. The result deals only with systems with finitely many peers to avoid having to extend Datalog to infinitely many relations.

THEOREM 7. *Let* (I, Γ) *be a positive system with a finite number of peers which converges to* $(I^*, \Gamma^*, \widetilde{\Gamma}^*)$*, and let* M_{min} *be the unique minimal model of the Datalog program* $c(I, \Gamma)$*. Then*

$$M_{min} = \cup_p P^*_{p,d}(I^*(p))$$

where $P_{p,d}$ *is the set of fully local deductive rules in* $\widetilde{\Gamma}^*(p) \cup \cup_q \Gamma^*(q,p)$*.*

5.2 Strongly-stratified Webdamlog

With negation, convergence is not guaranteed in the general case as illustrated by the following example.

EXAMPLE 3. *Consider the program that is stratified in the sense of Datalog with stratified negation:*

intensional s@p, r@p, r@q
at p: r@q() :- r@p()
r@p() :- ¬s@p()
at q: r@p() :- r@q()
s@p() :-

Any run of this system that begins with p converges to a state where p delegates r@q():- to q and q delegates r@p():- and s@p():- to p. On the other hand, runs that begin with q converge to a state where p delegates nothing to q and q delegates s@p():- to p.

As already mentioned for the non-monotone updates in the previous subsection, one may adapt methods of view maintenance to solve the problem. We develop in this section an alternative in which syntactic restrictions prohibit circles of false deductions, without having to deal with the complexity of view maintenance in presence of belief revision. Note that most of the examples of the paper belong to (or are easily adapted to) this restricted class.

A *stratification* σ' is an assignment of numbers to relations, i.e., to pairs $r@p$. If $\sigma'(r@p) = i$, we say that $r@p$ is

in the ith stratum. The stratification is *strong* if for each i, all the relations in the ith stratum refer to the same peer. Given a strong stratification σ', an instantiated rule is σ'-stratified if all relation names of positive body atoms appear in a stratum smaller or equal to that of the head relation and all relation names of negative terms belong to a strictly smaller stratum. Note that a stratification for Example 3 would not be strong because $r@p$ and $r@q$ have to be in the same stratum, although they belong to different peers.

In our setting, we see a strong stratification σ' of $\mathcal{I}$ as an extra component of the system's schema. The strong stratification works much like the typing constraint σ in that it tells us whether a particular rule instantiation is legal. Specifically, a peer is only allowed to use instantiated rules which are σ'-stratified. Observe that our use of stratification is in the spirit of classical Datalog with stratified negation, namely preventing cycling through negation. However, the way stratification is enforced is somewhat different. In the centralized context, one analyzes the program and checks for the existence of a stratification. In the distributed case, this is not possible because no one has access to the entire program. Also, the use of relation and peer variables makes such a computation even less conceivable. So, instead, one assumes that a stratification is imposed and the computation is such that it prevents deriving facts with rule instantiations that would violate the strong stratification.

There is a subtlety with strong stratification arising from general delegation. Indeed, we will see that the result does not hold for WL. So the next result deals simply with view delegation, i.e., the language VWL. One of the advantages of VWL is that at the time a rule is delegated, it is possible to check that it does not violate the strong stratification. We consider systems with finitely many peers, where the extensional facts are fixed and only the intensional delegations vary. Formally, a Webdamlog system is said to be *strongly-stratified* if for some strong stratification σ':

1. its local computation is constrained by the stratification σ'.

2. Each extensional relation $m@p$ is made persistent with a rule of the form $m@p(\overline{U})$:- $m@p(\overline{U})$ and these are the only active rules in the system[1]. We say the system is *purely intensional*.

Observe that, by Condition (2), the set of extensional facts is fixed whereas it was increasing for positive systems. So Condition (2) here is more restrictive than for positive systems. Thus, strictly speaking the two classes are incomparable. Clearly, it would be interesting to consider classes that would include both.

We are now ready to present our results, following the same logic as in the previous section.

THEOREM 8 (CONVERGENCE). *All strongly-stratified VWL systems over a finite number of peers converge.*

This result does not hold if we allow general delegation instead of view delegation. This is because with general delegation, a peer p may delegate a partially instantiated rule to q. As the relation and peer terms of the rule may contain variables, peer p may not be able to decide whether the rule is σ'-stratified, and neither will q (or later peers) as they do not know which relations p used to launch the delegation. So enforcement of the stratification is not straightforward. This is illustrated by the following example.

EXAMPLE 4. *Consider the following program:*

intensional m@p, s@q, r@q
at p: m@p($x) :- m@p($x), r@q($x)
m@p($x) :- r@q($x), ¬s@q()
at p': s@q() :-
at q: r@q(a) :-

Consider a run that starts by firing p, q, then p. Then the rule m@p(a):- is delegated by q to p and will remain forever. Now, consider a run that starts by firing p'. Then q will know s@q():-. from the beginning and will never delegate m@p(a):-.

Convergence also holds for strongly-stratified VWL systems in the presence of insertions as well as deletions.

THEOREM 9 (UPDATE). *Let (I,Γ) be a VWL system with strong stratification σ' over a finite number of peers. Consider (I^+,I^-, Γ^+,Γ^-) where I^+, I^- are sets of extensional facts and Γ^+,Γ^- are sets of deductive rules. For each run of the system (I,Γ), if for some k a given state (I_k,Γ_k, $\widetilde{\Gamma}_k$) is replaced by ($I_k \cup I^+ \setminus I^-$, $\Gamma_k \cup \Gamma^+ \setminus \Gamma^-, \widetilde{\Gamma}_k$), then the modified run converges to the convergence state of the σ'-stratified system ($I \cup I^+ \setminus I^-$, $\Gamma \cup \Gamma^+ \setminus \Gamma^-$).*

This theorem can obviously be generalized to any sequence of updates. The final theorem of this section shows that the set of facts computed by a σ'-stratified system corresponds to the set of facts in the minimal model of a centralized version of the system. As in the previous section, we associate a σ'-stratified Webdamlog system (I, Γ) with the set $\cup_p(I(p) \cup \Gamma(p))$ composed of the facts and rules of all peers. We then transform this set of facts and rules into a standard Datalog program by instantiating the variable predicates in the rules and removing rules which violate the typing constraints σ or the strong stratification σ'. We use $c_s(I, \Gamma)$ to refer to the resulting Datalog program.

THEOREM 10 (CENTRALIZED). *Let (I, Γ) be a σ'-stratified system with a finite number of peers and rules in SWL, which converges to $(I^*, \Gamma^*, \widetilde{\Gamma}^*)$, and let M_{min} be the unique minimal model of the Datalog program $c_s(I, \Gamma)$. Then*

$$M_{min} = \cup_p P^*_{p,d}(I^*(p))$$

where $P^*_{p,d}$ is the set of fully local deductive rules in $\widetilde{\Gamma}^*(p) \cup \cup_q \Gamma^*(q, p)$.

6. SYSTEM AND OPTIMIZATION

In this section, we briefly mention the system that motivated the present work and standard optimization techniques that make the approach feasible.

Webdam Exchange. In Webdam Exchange [5], data (XML documents, collections), access rights, secrets, localization, and knowledge about other peers, are all seen as logical statements. These statements can be communicated, replicated, queried, and updated, while keeping track of time

[1]Technically speaking, if we want to use variable or peer relations in the rule heads, then we must forbid instantiations which yield extensional relations in the heads.

and provenance. Localization statements guide the search for pieces of information to access or update. The same information may be supported by different peers. Each statement carries its own access control enforcement. It may be *authenticated*, in the sense that it is possible to verify the identity of the participant who created the statement and that this participant was indeed entitled to create it. Information content may also be *protected* in the sense that only participants with a particular secret (e.g., a decryption key or a login/password pair) can read it.

We have implemented a proof-of-concept system that handles all the components of the knowledge base. We also implemented a lighter system designed for smartphones. First versions of the systems are up and running and will be demonstrated at [10]. The entire system may be seen as a distributed knowledge base. Most of the effort so far has been dedicated to the local management of access rights and the verification of data. The reasoning to find data is hardwired in Java programs. We intend to replace these programs by some reasoning using the Webdamlog system we started implementing. In the new setting, each peer has its own logic (a set of rules) that depends on the particular application and on its resources and may acquire new rules by update or by delegation.

Optimization. To render the approach feasible, we have to rely intensively on some known optimization techniques. We briefly mention them next and see how they fit in the Webdamlog picture.

(Differential technique) Consider a peer p who has the rule $s@q(x, y)$:- $r@p(x, y)$ with $s@q$ an extensional relation. Suppose that $r@p$ is a very large relation that changes infrequently. Each time we visit p we have to send to q the current version of $r@p$, say a set K_n of tuples. This is a clear waste of communication resources. It is preferable to send the symmetric difference of $r@p$, i.e., send a set of updates Δ with the semantics that $K_n = \Delta(K_{n-1})$, since q already knows K_{n-1}. If $s@q$ is intensional, we face a similar issue; it is preferable to send the new set of delegation rules as Δ rather than sending the entire set.

(Seed-based delegation) Consider again the rule:

at p: m@q() :- m_1@p($x), m_2@p'($x)

Now suppose that $m_1@p(a_i)$ holds for $i = [1..1000]$. We need to install 1000 rules. However, in this particular case, we can install a single rule at p' and send many facts:

at p': m@q() :- $seed_{r,1,p}$@p'($x), m_2@p'($x)
at p': $seed_{r,1,p}$@p'(a_i). (for each i)

Note that it now becomes natural to use a differential technique to maintain delegation. In particular, if the delegation from p to q does not change, there is no need to send anything. If it does, one needs only to send the delta on $seed_{r,1,p}@p'$. Observe that we have replaced the task of installing and uninstalling delegation rules by that of sending insertion and deletion messages in a persistent extensional (seed) relation that controls a rule.

(Query-subquery and delegation) Consider the following example of a rule in BIP (Bob's IPhone), where photosAlice@BIP is intensional:

at BIP: photosAlice@BIP($X,$Y) :-
photos@picasa(Alice,$X,$Y)

This rule says that to find the photos of Alice, one needs to ask Picasa. The formal semantics says that we install the following [*Upload*] rule at Picasa:

at Picasa: photosAlice@BIP($X,$Y) :-
photos@picasa(Alice,$X,$Y)

which will result in uploading in BIP all the photos. However, observe that this has no effect on the state since *photosAlice* is only intensional. This uploading may therefore be considered a waste of resources. An optimizer may decide not to install the [*Upload*] rule at Picasa, i.e., not ask Picasa to upload anything. Now suppose that Bob asks his Iphone for the photos of Sue:

query@BIP($X) :- photosAlice@BIP($X, Sue)

where *query* is an extensional predicate. Now obtaining photos from Picasa changes the state. So the optimizer will install on Picasa the rule:

at Picasa: photosAlice@BIP($X,Sue) :-
photos@picasa(Alice,$X,Sue)

Observe that the optimizer performed some form of resolution in the spirit of query-subquery [28] or rewriting in the Magic Set style [11] (see also [6]). Indeed, the entire management of delegation can be optimized using these techniques. Note that strictly speaking this may change the semantics of applications: the derivation of some facts may take a little longer than if we had installed all the delegations in advance.

7. CONCLUSION AND FUTURE WORK

We have introduced a new Datalog-style language for distributed data management. The main novelty is the notion of delegation that allows a peer to install rules at other peers. We have studied the expressivity of the language and of restrictions. We have also studied convergence properties for fragments of the language.

One should observe that the power of delegation critically depends on the exact definition of the model. The situation would be different, for instance, if we were to consider an asynchronous version of the model in which messages between peers are not instantaneous. A natural direction for future work is the extension of our study of the power of delegation and related issues (e.g. possibility of electing a leader) to different variants of the model.

We use in our model a rather strict notion of type based on fixed arity and fixed sort for each column. It would be interesting to be more tolerant and allow some form of polymorphism (e.g., relations with arbitrary arity) or recursive types (an email may include an email). Note that this is useful for the exchange of XML data (and our implementation in Webdam Exchange is based on XML).

The notion of provenance (also important for Webdam Exchange) is not considered in the model we presented here. We intend to study it and in particular to develop methods for tracing the origin of deduced facts.

As another possible direction for future work, Active XML considers intensional data of a very different form, namely functions that may be included in documents and are defined intensionally. It would be interesting to investigate the relationships between these two kinds of intensional data.

Acknowledgements. We thank David Gross-Amblard (Univ. Bourgogne), Yannis Katsis and Bruno Marnette (INRIA Saclay), Philippe Rigaux (CNAM, Paris), Marie-Christine Rousset (Univ. Grenoble) and Victor Vianu (UCSD) for discussions on this work.

8. REFERENCES

[1] 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.

[2] S. Abiteboul, O. Benjelloun, and T. Milo. Positive active xml. In *PODS*, pages 35–45, 2004.

[3] S. Abiteboul, O. Benjelloun, and T. Milo. The active xml project: an overview. *The VLDB Journal*, 17:1019–1040, 2008.

[4] S. Abiteboul, M. Bienvenu, A. Galland, and M.-C. Rousset. Distributed datalog revisited. In Gottlob [15].

[5] S. Abiteboul, A. Galland, and A. Polyzotis. Web information management with access control. In preparation, 2011.

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

[7] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *PODS*, pages 221–230, 2008.

[8] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[9] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.

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

[11] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.

[12] A. K. Chandra and D. Harel. Horn clauses queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[13] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 410(2-3):168–201, 2009.

[14] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[15] G. Gottlob, editor. *Datalog 2.0*, LNCS. Springer, to appear.

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

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

[18] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *VLDB*, pages 87–96, 1989.

[19] G. Lausen, B. Ludäscher, and W. May. *On Active Deductive Databases: The Statelog Approach*, volume 1472 of *Lecture Notes in Computer Science*, pages 69–106. Birkhauser Basel, 1998.

[20] C. Liu, Y. Mao, M. Oprea, P. Basu, and B. T. Loo. A declarative perspective on adaptive manet routing. In *PRESTO*, pages 63–68, 2008.

[21] B. T. Loo, T. Condie, M. 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.

[22] 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. *Communications of the ACM*, 52(11):87–95, 2009.

[23] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Operating Systems Review*, 39:75–90, 2005.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, 2005.

[25] J. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In A. Gill and T. Swift, editors, *Practical Aspects of Declarative Languages*, volume 5418 of *LNCS*, pages 76–90. Springer, 2009.

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

[27] T. C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

[28] L. Vieille. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. Expert Database Systems*, pages 179–193, 1986.

[29] ERC grant Webdam. http://webdam.inria.fr.