Maintaining Distributed Logic Programs Incrementally

Vivek Nigam
LMU, GERMANY
vivek.nigam@ifi.lmu.de

Limin Jia
CMU, USA
liminjia@cmu.edu

Boon Thau Loo
UPENN, USA
boonloo@cis.upenn.edu

Andre Scedrov
UPENN, USA
scedrov@math.upenn.edu

Abstract

Distributed logic programming languages, that allow both facts and programs to be distributed among different nodes in a network, have been recently proposed and used to declaratively program a wide-range of distributed systems, such as network protocols and multi-agent systems. However, the distributed nature of the underlying systems poses serious challenges to developing efficient and correct algorithms for evaluating these programs. This paper proposes an efficient asynchronous algorithm to compute incrementally the changes to the states in response to insertions and deletions of base facts. Our algorithm is formally proven to be correct in the presence of message reordering in the system. To our knowledge, this is the first formal proof of correctness for such an algorithm.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Operational Semantics

General Terms Algorithms, Theory, Correctness

Keywords Distributed Datalog, Logic Programming, Incremental Maintenance

1. Introduction

One of the most exciting developments in computer science in recent years is that computing has become increasingly distributed. Both resources and computation no longer reside in a single place. Resources can be stored in different machines possibly around the world, and computation can also be performed by different machines, *e.g.* cloud computing. Since machines usually run asynchronously and under very different environments, programming computer artifacts in such frameworks has become increasingly difficult as programs have to be at the same time correct, readable, efficient and portable. There has therefore been a recent return to using declarative programming languages, based on Prolog and Datalog, to program distributed systems such as networks and multi-agent robotic systems, *e.g.* Network Datalog (*NDlog*) [10], MELD [5], Netlog [6], DAHL [11], Dedalus [4]. When programming in these declarative languages, programmers usually do not need to specify *how* computation is done, but rather *what* is to be computed. Therefore declarative programs tend to be more readable, portable, and orders of magnitude smaller than their imperative counterpart.

Distributed systems, such as networking and multi-agent robotic systems, deal at their core with maintaining states by allowing each node (agent) to compute locally and then propagate its local states to other nodes in the system. For instance, in routing protocols, at each iteration each node computes locally its routing tables based on information it has gained so far, then distributes the set of derived facts to its neighbors. We can specify these systems as distributed logic programs, where the base facts as well as the rules are distributed among different nodes in the network.

Similarly to its centralized counterpart, one of the main challenges of implementing these distributed logic programs is to efficiently and correctly update them when the base facts change. For distributed systems, the communication costs due to updates also need to be taken consideration. For instance, in the network setting, when a new link in the network has been established or an old link has been broken, the set of derived routes need to be updated to reflect the changes in the base facts. It is impractical to recompute each node's state from-scratch when changes occur, since that would require all nodes to exchange their local states including those that have been previously propagated. For example, in the path-vector protocol used in Internet routing, recomputation from-scratch would require all nodes to exchange all routing information.

A better approach is to maintain the state of distributed logic programs incrementally. Instead of reconstructing the entire state, one only modifies previously derived facts that are affected by the changes of the base facts, while the remaining facts are left untouched. For typical network topologies, whenever a link update happens, incremental recomputation requires less bandwidth and results in much faster protocol convergence times when compared to recomputing a protocol from scratch.

This paper develops algorithms for incrementally maintaining recursive logic programs in a distributed setting. Our algorithms allow asynchronous execution among agents. No agent needs to *stop* computing because some other agent has not concluded its computation. Synchronization requires extra communication between agents, which comes at a huge performance penalty. In addition, we also allow update messages to be received out of order. We do not assume the existence of a *coordinator* in the system, which matches the realty of distributed systems. Finally, we develop techniques that ensure the termination of updates even in the presence of recursive logic programs.

PDPP'11, July 20–22, 2011, Odense, Denmark.

More concretely, we propose an asynchronous incremental logic programming maintenance algorithm, based on the *pipelined semi-naïve* (PSN) evaluation strategy proposed by Loo *et al.* [10]. PSN relaxes the traditional semi-naïve (SN) evaluation strategy for Datalog by allowing an agent to change its local state by following a local pipeline of update messages. These messages specify the insertions and deletions scheduled to be performed to the agents' local state. When an update is processed, new updates may be generated and those that have to be processed by other agents of the system are transmitted accordingly.

We discovered that existing PSN algorithms [9, 10] may produce incorrect results if the messages are received out of order. We formally prove the correctness of our PSN algorithm, which is lacking from existing work. What makes the problem hard is that we need to show that, in a distributed, asynchronous setting, the state computed by our algorithm is correct regardless of the order in which updates are processed. Unlike prior PSN proposals [9, 10], our algorithm does not require that message channels be FIFO, which is for many distributed systems an unrealistic assumption.

Guaranteeing termination is another challenge for developing an incremental maintenance algorithm for distributed recursive logic programs. Typically, in a centralized synchronous setting, algorithms, such as DRed [7], guarantee the termination of updates caused by insertion by maintaining the set of derivable facts, and discarding new derivations of previously derived facts. However, to handle updates caused by deletion properly, DRed [7] needs to first delete facts caused by deletion of base facts, then re-derive any deleted fact that has an alternative derivation. Re-derivation incurs communication costs, which degrade the performance in a distributed setting. This argues for maintaining the multiset of derivable facts, where no re-derivation of facts is needed, since nodes keep track of all possible derivations for any fact. However, termination is no longer guaranteed, as cycles in the derivation of recursive programs allow facts to be supported by infinitely many derivations.

To tackle this problem, we adapt an existing centralized solution [12] to distributed settings. For any given fact, we add annotations containing the set of base and intermediate facts used to derive that fact. These per-fact annotations are then used to detect cycles in derivations. We formally prove that in a distributed setting, the annotations are enough to detect when facts are supported by infinitely many derivations and guarantee termination of our algorithm.

This paper makes the following technical contributions, after introducing some basic definitions in Section 2:

- We propose a new PSN-algorithm to maintain distributed logic programs incrementally (Section 3). This algorithm only deals with distributed non-recursive logic programs. (Recursive programs is dealt in Section 5.)
- We formally prove that PSN is correct (Section 4). Instead of directly proving PSN maintains distributed logic programs correctly, we construct our proofs in two steps. First, we define a synchronous algorithm based on SN evaluations, and prove the synchronous SN algorithm is correct. Then, we show that any PSN execution computes the same result as the synchronous SN algorithm.
- We extend the basic algorithm by annotating each fact with information about its derivation to ensure the termination of maintaining distributed states (Section 5), and prove its correctness.
- We point out the limitations of existing maintenance algorithms in a distributed setting where channels are not necessarily FIFO (Section 6) and comment on related work (Section 7);

Finally, we conclude with some final remarks in Section 8. All proofs appear in the companion technical report [14].

2. Distributed Datalog

We present *Distributed Datalog* (*DDlog*), which extends Datalog programs by allowing Datalog rules to be distributed among different nodes. *DDlog* is the core sublanguage common to many of the distributed Datalog languages, such as *NDlog* [10], MELD [5], Netlog [6], and Dedalus [4]. Our algorithms maintain the states for *DDlog* programs.

2.1 Syntax and Evaluation

Syntax. Similar to Datalog programs, a *DDlog* program consists of a (finite) set of logic rules of the form $h(\vec{t})$ `:-` $b_1(\vec{t}_1), \ldots, b_n(\vec{t}_n)$, where the commas are interpreted as conjunctions and the symbol `:-` as reverse implication. Following [16], we assume a *finite* signature of predicate and constant symbols, but no function symbols. A *fact* is a ground atomic formula. For the rest of this paper, we use fact and predicate interchangeably.

We say that a predicate p depends on q if there is a rule where p appears in its head and q in its body. The *dependency graph* of a program is the transitive closure of the dependency relation using its rules. We say that a program is (*non*)*recursive* if there are (no) cycles in its dependency graph. We classify the predicates that do not depend on any predicates as base predicates (facts), and the remaining predicates as derived predicates.

To allow distributed computation, *DDlog* extends Datalog by augmenting its syntax with the location operator `@` [10], which specifies the location of a fact. The following *DDlog* program computes the reachability relation among nodes:

```
r1: reachable(@S,D) :- link(@S,D).
r2: reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
```

It takes as input `link(@S,D)` facts, each of which represents an edge from the node itself (`S`) to one of its neighbors (`D`). The location operator `@` specifies where facts are stored. For example, `link` facts are stored based on the value of the `S` attribute.

Distributed Evaluation. Rules `r1-r2` recursively derive `reachable(@S,D)` facts, each of which states that the node `S` is reachable from the node `D`. Rule `r1` computes one-hop reachability, given the neighbor set of `S` stored in `link(@S,D)`. Rule `r2` computes transitive reachability as follows: if there exists a link from `S` to `Z`, and the node `D` is reachable from `Z`, then `S` can also reach `D`.

In a distributed setting, initially, each node in the system stores the link facts that are relevant to its own state. For example, the fact `link(@2,4)` is stored at the node `2`. To compute all reachability relations, each node runs the exact same copy of the program above concurrently. Newly derived facts may need to be sent to the corresponding nodes as specified by the `@` operator.

Rule localization. As illustrated by the rule `r2`, the atomic formulas in the body of the rules can have different location specifiers indicating that they are stored on different nodes. To apply such a rule, facts may need to be gathered from several nodes, possibly different from where the rule resides. To have a clearly defined semantics of the program, we apply *rule localization* rewrite procedure as shown in [10] to make such communication explicit. The *rule localization* rewrite procedure transforms a program into an equivalent one (called *localized* program) where all elements in the body of a rule are located at the same location, but the head of the rule may reside at a different location than the body atoms. This procedure improves performance by eliminating the need of unnecessary communication among nodes, as a node only needs the facts locally stored to derive a new fact. For example, the followings two rules are the localized version of `r2`:

```
r2-1: reachable(@S,D) :- link(@S,Z), aux(@S,Z,D).
r2-2: aux(@S,Z,D) :- reachable(@Z,D), co-link(@Z,S).
```

Here, the predicate `aux` is a new predicate: it does not appear in the original alphabet of predicates and the fact `co-link(@Z,S)` is true if and only if `link(@S,Z)` is true. The predicate `co-link(@Z,S)` is used to denote that the node `Z` knows that the node `S` is one of its neighbors. As specified in the rule `r2-1`, these predicates are used to inform all neighbors, `S`, of node `Z` that the node `Z` can reach node `D`. It is not hard to show, by induction on the height of derivations, that this program is equivalent to the previous one in the sense that a `reachable` fact is derivable using one program if and only if it is derivable using the other. For the rest of this paper, we assume that such localization rewrite has been performed.

2.2 Multiset Semantics

The semantics of *DDlog* programs is defined in terms of the (multi)set of derivable facts (*least model*). We call such a (multi)set, the *state* of the program. In database community, it is called the *materialized view* of the program. For instance, in the following non-recursive program, `p`, `s`, and `t` are derived predicates and `u`, `q`, and `r` are base predicates.

```
{p :- s,t,r;  s :- q;  t :- u;  q :-;  u :-}.
```

The (multi)set of all the ground atoms that are derivable from this program, is {`s, t, q, u`}. For this example, each fact is supported by only one derivation and therefore the same state is obtained whether the state is the set, or the multiset of derivable facts. If we add, the rule `s :- u` to this program, then the state when using the multiset semantics of the resulting program would change to {`s, s, t, q, u`} where `s` appears twice. This is because there are two different ways to derive `s`: one by using `q` and the other by using `u`. Our choice of multiset-semantics is essential for correctness, which we discuss in detail in Section 6.

2.3 Incremental State Maintenance

Changes to the base predicates of a *DDlog* program will change its state. The goal of this paper is to develop a correct asynchronous algorithm that incrementally maintains the state of *DDlog* programs as updates occur in the system. The main idea of the algorithm is to first compute only the changes caused by the updates to the base predicates, then apply the changes to the state. For instance, when a base fact is inserted, the algorithm computes all the facts that were not in the state before the insertion, but are now derivable. Similarly, when a deletion occurs, the algorithm computes all the facts that were in the state before the deletion, but need to be removed. We introduce notations for defining such an algorithm here, and we formally define our algorithms and prove them correct in the next few sections starting from Section 3.

We denote an update as a pair $\langle U, p(\vec{t})\rangle$, where U is either `+`, denoting an insertion, or `-`, denoting a deletion, and $p(\vec{t})$ is a ground fact. We call an update of the form $\langle$`+`$, p(\vec{t})\rangle$ an *insertion update*; and $\langle$`-`$, p(\vec{t})\rangle$ a *deletion update*. We write $\mathcal{U}$ to denote a multiset of updates. For instance, the following multiset of updates

$$\mathcal{U} = \{\langle +, q(@1, d)\rangle, \langle -, q(@2, a)\rangle, \langle -, q(@2, a)\rangle\},$$

specifies that two copies of the fact $q(@2, a)$ should be deleted from node 2's state, while one copy of the fact $q(@1, d)$ should be inserted into node 1's state.

We use $\uplus$ as the multiset union operator, and $\setminus$ as the multiset minus operator. We write P to denote the multiset of ground atoms of the form $p(\vec{t})$ (atoms whose predicate name is p), and ΔP to denote the multiset of updates to predicate p. We write P^ν to denote the updated multiset of predicate p based on ΔP. P^ν can be computed from P and ΔP by union P with all the facts inserted by ΔP and minus the facts deleted by ΔP. For ease of presentation, we use the predicate name Δp in places where we need to use the updates, and p^ν in places where we need to use the updated multiset. For instance, if the multiset of q is $\{q(a), q(a), q(b), q(c)\}$ and we update it with $\mathcal{U}$ shown above, the resulting multiset (Q^ν) for q^ν is $\{q(b), q(c), q(d)\}$.

Rules for computing updates. The main idea of computing updates of a *DDlog* program given a multiset of updates to its base predicates is that we can modify the rules in the corresponding program to do so. Consider, for example, the rule p `:-` b_1, b_2 whose body contains two elements. There are the following three possible cases that one needs to consider in order to compute the changes to the predicate p: Δp `:-` $\Delta b_1, b_2$, Δp `:-` $b_1, \Delta b_2$, and Δp `:-` $\Delta b_1, \Delta b_2$. The first two just take into consideration the changes to the predicates b_1 and b_2 alone, while the last rule uses their combination. We call these rules *delta-rules*.

Following [1, 16], we can simplify the delta-rules above by using the state of p^ν, as defined above. The delta-rules above are changed to Δp `:-` $\Delta b_1, b_2$ and Δp `:-` $b_1^\nu, \Delta b_2$, where the second clause encompasses all updates generated by changes to new updates in both b_1 and b_2 as well as only changes to b_2.

Generalizing the notion of delta-rules described above, for each rule $h(\vec{t})$ `:-` $b_1(\vec{t}_1), \ldots, b_n(\vec{t}_n)$ in a program, we create the following delta insertion and deletion rules, where $1 \leq i \leq n$:

$$\langle +, h(\vec{t})\rangle : -b_1^\nu(\vec{t}_1), \ldots, b_{i-1}^\nu(\vec{t}_{i-1}), \Delta b_i(\vec{t}_i), b_{i+1}(\vec{t}_{i+1}), \ldots, b_n(\vec{t}_n)$$
$$\langle -, h(\vec{t})\rangle : -b_1^\nu(\vec{t}_1), \ldots, b_{i-1}^\nu(\vec{t}_{i-1}), \Delta b_i(\vec{t}_i), b_{i+1}(\vec{t}_{i+1}), \ldots, b_n(\vec{t}_n)$$

The first rule applies when Δb_i is an insertion, and the second one applies when Δb_i is a deletion.

By distinguishing predicates with ν and without ν one does not derive the same derivation twice [7].

3. Basic PSN Algorithm for Non recursive Programs

We first present an algorithm for incremental maintenance of distributed non-recursive logic programs. We do not consider termination issues in the presence of recursive programs, which allows us to focus on proving the correctness of pipelined execution. In Section 5, we will present an improved algorithm that provably ensures termination of recursive programs.

3.1 System Assumptions

Our model of distributed systems makes two main assumptions, which are realistic for many systems, such as in networking and systems involving robots.

The first assumption, following [10], is the *bursty model*: once a burst of updates is generated, the system eventually *quiesces* (does not change) for a time long enough for all the nodes to reach a fixed point. Without the bursty model, the links in a network could be changing constantly. Due to network propagation delays, no routing protocol would be able to update routing tables to correctly reflect the latest state of the network. Similarly, if the environment where a robot is situated changes too quickly, then the robot's internal knowledge of the world would not be useful for it to construct a successful plan. The bursty model can be seen as a compromise between completely synchronized models of communication and completely asynchronous models.

The second assumption is that messages are never lost during transmission. Here, we are not interested in the mechanisms of message transmission, but we assume that any message is eventually received by the correct node specified by the location specifier `@`. Differently from previous work [9, 10], it is possible for messages to be reordered in our model. We do not assume that a message that is sent before another message has to necessarily arrive at its destination first. There are existing protocols which acknowledge when messages are received and have the source nodes resend the messages in the event of acknowledgments timeouts, hence en-

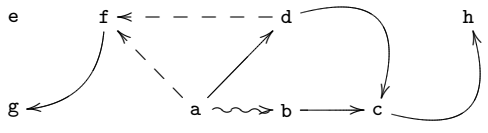

Figure 1. A simple network topology. A dashed arrow indicates an edge that is inserted, while a curly arrow an edge that is deleted. For instance, the edge from `d` to `f` is added, while the edge from `a` to `b` is deleted.

forcing that messages are not lost. Message reordering manifests itself in several practical scenarios. For instance, in addition to reordering of messages buffered at the network layer, network measurements studies have shown that packets may traverse different Internet paths for any two routers due to ISP policies [15]. In a highly disconnected environment such as in Robotics [5], messages from a given source to destination may traverse different paths due to available network connectivity during the point of transmission of each message.

3.2 PSN Algorithm

We propose Algorithm 1 for maintaining incrementally distributed states given a *DDlog* program. Algorithm 1 enhances the original pipelined evaluation strategy [10]. Since all facts are stored according to the `@` operator, we can use a single multiset $\mathcal{K}$ containing the union of states of all the nodes in the system. It is clear from the `@` operator where the data is stored. Similarly, we use a single multiset of updates $\mathcal{U}$ containing the updates that are in the system, but that have not yet been processed by any node.

Algorithm 1 starts with a multiset of updates $\mathcal{U}$ and the multiset $\mathcal{K}$ containing two copies of the state of all nodes in the system, one marked with ν and another without ν (see Section 2.3). The execution of one node of the system is specified by one iteration of the while-loop in Algorithm 1. In line 2, an update is *picked non-deterministically* from $\mathcal{U}$ to be processed next. However, only deletion updates whose corresponding facts are present in $\mathcal{K}$ are allowed to be picked. This is specified by the operation $removeElement(\mathcal{K})$, which avoids facts to have negative counts. Once an update is picked, the ν table is updated according to the type of update in lines 3–6. In lines 7–12, the picked update is used to *fire* delta-rules and create new updates that are then inserted into the multiset $\mathcal{U}$ (lines 13–15). This last step intuitively corresponds to a node sending new messages to other nodes, even to itself. Finally in the remaining lines, the changes to the state without ν are *committed* according to the update picked, making the table with ν and without ν have the same elements again and ready for the execution of the next iteration.

We prove that Algorithm 1 terminates for non-recursive *DDlog* programs.

LEMMA 1. *For non-recursive* DDlog *programs, PSN executions always terminate.*

The idea behind the proof is that since the dependency graph of non-recursive programs is a DAG (does not have cycles), whenever an update is picked and used to fire delta-rule, all updates created involve facts whose predicate names appear necessarily in a position "higher" in the dependency graph. Eventually, the set of updates will be empty since the dependency graph has a bounded height. Thus, the algorithm finishes. This argument is valid regardless of the order in which updates are picked.

An Example Execution. We illustrate an execution of Algorithm 1 using the topology in Figure 1 and the following program adapted from [7], which specifies two and three hop reachability:[1]

```
hop(@X,Y) :- link(@X,Z), link(@Z,Y)
tri_hop(@X,Y) :- hop(@X,Z), link(@Z,Y)
```

Here the only base predicate is `link`. Furthermore, assume that the state is as given below, where we elide the `@` symbols. For example, the facts `link(@a,b)` and `hop(@a,c)` are in the state. Also at the beginning, the multiset of predicates with ν is the same as the multiset of predicates without ν, so we elide the former.

```
Link = {link(a,b), link(a,d), link(d,c), link(b,c),
        link(c,h), link(f,g)}
Hop = {hop(a,c), hop(a,c), hop(d,h), hop(b,h)}
Tri_hop = {tri_hop(a,h), tri_hop(a,h)}
```

In the state above some facts appear with multiplicity greater than one, which means that there are more than one derivation supporting such facts. Assume as depicted in Figure 1 that there is the following changes to the set of base facts `link`:

$\mathcal{U}$ = {⟨+, link(d,f)⟩, ⟨+, link(a,f)⟩, ⟨-, link(a,b)⟩}

Algorithm 1 first *picks* an update non-deterministically, for instance, the update $u = \langle +, \texttt{link(a,f)} \rangle$, which causes an insertion of the fact `link(a,f)` to the table marked with ν. Now $\texttt{Link}^\nu$ is as follows:

$$\texttt{Link}^\nu = \{\texttt{link}^\nu\texttt{(a,b)}, \texttt{link}^\nu\texttt{(a,d)}, \texttt{link}^\nu\texttt{(d,c)}, \texttt{link}^\nu\texttt{(b,c)}, \texttt{link}^\nu\texttt{(c,h)}, \texttt{link}^\nu\texttt{(f,g)}, \texttt{link}^\nu\texttt{(a,f)}\}$$

Then, u is used to propagate new updates by *firing* rules, which creates a single insertion update: $\langle +, \texttt{hop(a,g)} \rangle$. Finally, the change due to the update u is committed to the table without ν. The new multiset of updates and the new multiset of the `link` facts are as follows:

$\mathcal{U} = \{\langle +, \texttt{hop(a,g)}\ \rangle, \langle +, \texttt{link(d,f)} \rangle, \langle -, \texttt{link(a,b)} \rangle\}$

```
Link = {link(a,b), link(a,d), link(d,c), link(b,c),
        link(c,h), link(f,g), link(a,f)}
```

Asynchronous Execution. As previously mentioned, in a distributed setting, agents need to run as asynchronously as possible,

Algorithm 1 Basic pipelined semi-naïve algorithm.

```
1:  while U.size > 0 do
2:      δ ← U.removeElement(K)
3:      if δ is an insertion update ⟨+, p(t⃗)⟩
4:          P^ν = P ⊎ {p(t⃗)}
5:      if δ is a deletion update ⟨-, p(t⃗)⟩
6:          P^ν = P \ {p(t⃗)}
7:      if δ is an insertion update ⟨+, b(t⃗)⟩
8:          execute all insertions delta-rules for b:
9:              ⟨+, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
10:     if δ is a deletion update ⟨-, b(t⃗)⟩
11:         execute all deletion delta-rules for b:
12:             ⟨-, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
13:     for all derived insertion (deletion) updates u do
14:         U.insert(u)
15:     end for
16:     if δ is an insertion update ⟨+, p(t⃗)⟩
17:         P = P ⊎ {p(t⃗)}
18:     if δ is a deletion update ⟨-, p(t⃗)⟩
19:         P = P \ {p(t⃗)}
20: end while
```

[1] Technically, the given program passes first through the rule localization procedure described in Section 2. However, for the purpose of illustration, we use instead this un-localized program.

since synchronization among agents involves undesired communication overhead.

Synchronized algorithms proposed in the literature admit the following invariant: in an iteration one only processes updates that insert or delete facts that are supported by derivations of some specific height. This is no longer the case for Algorithm 1: it picks updates non-deterministically. In the example above, one does not necessarily process all the updates involving `link` facts before processing `hop` or `tri_hop` facts. In fact, in the next iteration of Algorithm 1, a node is allowed to pick the update $\langle +, \texttt{hop(d,g)} \rangle$ although there are insertions and deletions of `link` facts still to be processed. However, this asynchronous behavior makes the correctness proof for Algorithm 1 much harder and forces us to proceed our correctness proofs quite differently.

Algorithm 1 sequentializes the execution of all nodes: in each iteration of the outermost while loop, one node picks an update in its queue, fires all the delta-rules and commits the changes to the state, while other nodes are idle. However this is only for the convenience of constructing the proofs of correctness. In a real implementation, nodes run Algorithm 1 concurrently. The correctness of this simplification is justified by Theorem 2 below. Intuitively, the localization procedure described in Section 2 ensures that all the predicates in the body are stored at the same location, which implies that updates on two different nodes can proceed independently, based only on their local states respectively.

Consider, as an illustrative example, the following localized program with two clauses:

(1) $p(@Y)$:- $s(@X, Y)$

(2) $s(@Y, X)$:- $q(@X), v(@X, Y)$.

Assume that there are two nodes n_1 and n_2 and that the initial state and set of updates are, respectively, $\{q(@n_1), v(@n_1, n_2)\}$ and $\{\langle +, s(@n_2, n_1)\rangle, \langle -, q(@n_1)\rangle\}$. If both nodes execute concurrently, then both updates are picked and used to fire the rules of the program. However, since the programs are localized, there is no need for the nodes n_1 and n_2 to communicate between each other during the execution of an iteration of Algorithm 1: they only need to access their own internal states. Node n_1 will fire a deletion delta-rule of rule (2) using the update $\langle -, q(@n_1)\rangle$ and the fact $v(@n_1, n_2)$, which are at node n_1. The update $\langle -, s(@n_2, n_1)\rangle$ is then created and sent to node n_2, while the fact $q(@n_1)$ is deleted from n_1's local state. Similarly, the node n_2 will fire an insertion delta-rule of rule (1) using the update $\langle +, s(@n_2, n_1)\rangle$ and creating the insertion update $\langle +, p(@n_1)\rangle$. Since the operations involved in the iterations do not interfere with each other, this concurrent execution can be replaced by a sequential execution where the node n_1 executes its iteration before the node n_2 and the resulting final state is the same.

For simplicity Theorem 2 only considers the case with two nodes running concurrently. The general case where more than two nodes running concurrently can be proved in a similar fashion.

THEOREM 2. *Let $\mathcal{P}$ be a localized* DDlog *program, and let W_I and $\mathcal{U}_I$ be an initial state and an initial multiset of updates. Let W_F and $\mathcal{U}_F$ be the state and the multiset of updates resulting from executing at different nodes two iterations, i_1 and i_2, of Algorithm 1 concurrently, where w.l.o.g. i_1 starts before or at the same time as i_2. Then the same state and multiset of updates, W_F and $\mathcal{U}_F$, are obtained after executing in a sequence i_1 and then i_2.*

4. Correctness of Basic PSN

The correctness proof relates the distributed PSN algorithm (Algorithm 1) to a synchronous SN algorithm (Algorithm 2), whose correctness is easier to show. After proving that Algorithm 2 is correct, we prove the correctness of Algorithm 1 by showing that an execution using distributed PSN can be transformed into an execution using SN.

4.1 Operational Semantics for Algorithm 1

To prove the correctness of Basic PSN, we first formally define the operational semantics of Algorithm 1 in terms of state transitions.

Algorithm 1 consists of three key operations: $pick$, $fire$ and $commit$. We call them basic commands, and an informal description are given below:

pick – A node picks non-deterministically one update, u, that is not a deletion of a fact that is not (yet) in the state, from the multiset of updates $\mathcal{U}$. If u is an insertion of predicate p, p^ν is inserted into the updated state P^ν; otherwise if it is a deletion update, p^ν is deleted from P^ν. This basic command is used in lines 2–6 in Algorithm 1.

fire – This command is used to execute all the delta-rules that contain Δp in their body, where $\langle U, p(\vec{t})\rangle$ has already been selected by the $pick$ command. After a rule is fired, the derived updates from firing this rule are added to the multiset $\mathcal{U}$ of updates. This basic command is used in lines 7–15 in Algorithm 1.

commit – Finally, after an update u has already been both picked and used to fire delta-rules, the change to the state caused by u is committed: if u is an insertion update of a fact p, p is inserted into the state P; otherwise, if it is a deletion update of p, p is deleted from the state P. This basic command is used in lines 16–19 in Algorithm 1.

A configuration s is a tuple $\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle$, where $\mathcal{K}$ is a multiset of facts, and $\mathcal{U}, \mathcal{P}$ and $\mathcal{E}$ are all multisets of updates. More specifically, at each iteration of the execution, $\mathcal{K}$ is a snapshot of the derivable facts, and it contains both the multiset (P) and the updated multiset (P^ν). The multiset $\mathcal{U}$ contains all the updates that are yet to be picked for processing; $\mathcal{P}$ contains the updates that have been picked and are scheduled to fire delta-rules; and finally $\mathcal{E}$ contains the updates that have been already used to fire delta-rules, but not yet committed into the state. At the end of the execution, $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{E}$ should be empty signaling that all updates have been processed, and $\mathcal{K}$ is the final state of the system.

The five functions depicted in Figure 2, that take a configuration and an update and return a new configuration, specify the semantics of the basic commands. The semantics of the $pick$ command is specified by $pick_I$, when the update is an insertion; and $pick_D$, when the update is a deletion. The $pick$ command moves, an update $\langle U, p(\vec{t})\rangle$ from $\mathcal{U}$ to $\mathcal{P}$, and updates the state in $\mathcal{K}$: $p^\nu(\vec{t})$ is inserted into $\mathcal{K}$ if U is +; it is deleted from $\mathcal{K}$ if U is -. Note that the rule $pick_D$ only applies when the predicate to be deleted actually exists in $\mathcal{K}$. Because messages may be re-ordered, it could happen that a deletion update message for predicate p arrives before p is derived based on some insertion updates. In an implementation, if such an update happens to be picked, we simply put it back to the update queue, and pick another update.

The rule $fire$ specifies the semantics of command $fire$, where we make use of the function $firRules$. This function takes an update, $\langle U, p(\vec{t})\rangle$, the current state, $\mathcal{K}$, and the set of rules, $\mathcal{R}$, as input and returns the multiset of all updates, $\mathcal{F}$, generated from firing all delta-rules that contain Δp in their body. The multiset $\mathcal{F}$ is then added to the multiset $\mathcal{U}$ of updates to be processed later.

Finally, the last two rules, $commit_I$ and $commit_D$, specify the operation of committing the changes to the state. Similar to the rules for $pick$, they either insert into or delete from the updated multiset P a fact $p(\vec{t})$.

A *computation run* of a program $\mathcal{R}$ is a valid sequence of applications of the functions defined in Figure 2. We call the first configuration of a computation run the initial configuration and its last configuration the resulting configuration.

- $pick_I(\mathcal{S}, \langle +, p(\vec{t})\rangle) = \langle \mathcal{K} \uplus \{p^\nu(\vec{t})\}, \mathcal{U} \setminus \{\langle +, p(\vec{t})\rangle\}, \mathcal{P} \uplus \{\langle +, p(\vec{t})\rangle\}, \mathcal{E}\rangle$, provided $\langle +, p(\vec{t})\rangle \in \mathcal{U}$.
- $pick_D(\mathcal{S}, \langle -, p(\vec{t})\rangle) = \langle \mathcal{K} \setminus \{p^\nu(\vec{t})\}, \mathcal{U} \setminus \{\langle -, p(\vec{t})\rangle\}, \mathcal{P} \uplus \{\langle -, p(\vec{t})\rangle\}, \mathcal{E}\rangle$, provided $\langle -, p(\vec{t})\rangle \in \mathcal{U}$ and $p^\nu(\vec{t}) \in \mathcal{K}$.
- $commit_I(\mathcal{S}, \langle +, p(\vec{t})\rangle) = \langle \mathcal{K} \uplus \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle +, p(\vec{t})\rangle\}\rangle$, provided $\langle +, p(\vec{t})\rangle \in \mathcal{E}$.
- $commit_D(\mathcal{S}, \langle -, p(\vec{t})\rangle) = \langle \mathcal{K} \setminus \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle -, p(\vec{t})\rangle\}\rangle$, provided $\langle -, p(\vec{t})\rangle \in \mathcal{E}$.
- $fire(\mathcal{S}, u) = \langle \mathcal{K}, \mathcal{U} \uplus \mathcal{F}, \mathcal{P} \setminus \{u\}, \mathcal{E} \uplus \{u\}\rangle$, provided $u \in \mathcal{P}$ and where $\mathcal{F} = firRules(u, \mathcal{K}, \mathcal{R})$.

Figure 2. Definition for the Basic Commands. Here $\mathcal{S}$ is the configuration $\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle$.

A single iteration of Algorithm 1, called *PSN-iteration*, is a sequence of these three commands. In particular, only one update is picked from $\mathcal{U}$ (lines 2–6), and used to fire delta-rules (lines 7–15), and then the change to the state (lines 16–19) is committed. For instance, in the example execution described in Section 3.2. The initial configuration is $\langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle$, where $\mathcal{K}$ and $\mathcal{U}$ are the same initial set of facts and updates shown in Section 3.2. Then the update $u = \langle +, \texttt{link(a,f)}\rangle$ from $\mathcal{U}$ is picked using the rule $pick_I$. The resulting configuration is the following, where the update u is moved to the set of picked updates:

$$\langle \mathcal{K} \uplus \{\texttt{link}^\nu\texttt{(a,f)}\}, \mathcal{U} \setminus \{u\}, \{u\}, \emptyset\rangle.$$

Then the *fire* rule is applied and creates the single update $u' = \langle +, \texttt{hop(a,g)}\rangle$, which is added to the set of updates, obtaining:

$$\langle \mathcal{K} \uplus \{\texttt{link}^\nu\texttt{(a,f)}\}, (\mathcal{U} \setminus \{u\}) \uplus \{u'\}, \emptyset, \{u\}\rangle.$$

Finally the *commit* rule is applied and the state is updated yielding:

$$\langle \mathcal{K} \uplus \{\texttt{link}^\nu\texttt{(a,f)}, \texttt{link(a,f)}\}, (\mathcal{U} \setminus \{u\}) \uplus \{u'\}, \emptyset, \emptyset\rangle.$$

which corresponds to the execution shown in Section 3.2, where the facts $\texttt{link}^\nu\texttt{(a,f)}$ and $\texttt{link(a,f)}$ are added, and the update u is removed from the original set of updates, while the propagated update u' is added to it.

The intuition above is formalized by using the more general notion of *complete-iterations*. Intuitively, a complete-iteration is a sequence of picks, fires and updates that use the same set of updates. A PSN-iteration is one special case of a complete-iteration where only one update is picked. In the example above the update used was $\langle +, \texttt{link(a,f)}\rangle$. A PSN execution is a sequence of PSN-iterations.

DEFINITION 3 (Complete-iteration).
A computation run is a complete-iteration *if it can be partitioned into a sequence of transitions using the pick commands ($pick_I$ and $pick_D$), followed by a sequence of transitions using the fire command, and finally a sequence of transitions using the commit command, such that the multiset of updates, $\mathcal{T}$, used by the sequence of $pick_I$ and $pick_D$ transitions is the same those used by the sequence of fire and those used by commit transitions.*

DEFINITION 4 (PSN-iteration). *A complete iteration is a* PSN-iteration *if the multiset of updates used by the pick commands contains only one update.*

DEFINITION 5 (PSN execution). *We call a computation run a PSN execution if it can be partitioned into a sequence of PSN-iterations, and in the last configuration $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{E}$ are empty.*

4.2 Correctness of SN Evaluations

We define an incremental maintenance algorithm based on synchronous semi-naïve (SN) evaluation. This algorithm itself is not practical for any real implementation because of high synchronization costs between nodes. We only use it as an intermediary step to prove the correctness of Algorithm 1.

Algorithm 2 Basic semi-naïve algorithm (multiset semantics).

```
1:  while U.size > 0 do
2:      for all insertion updates u = ⟨+, h(t⃗)⟩ in U do
3:          I_h.insert(h(t⃗))
4:      end for
5:      for all deletion updates u = ⟨-, h(t⃗)⟩ in U do
6:          D_h.insert(h(t⃗))
7:      end for
8:      for all predicates p do
9:          P^ν ← (P ⊎ I_p) \ D_p
10:     end for
11:     while U.size > 0 do
12:         δ ← U.removeElement(K)
13:         if δ is an insertion update ⟨+, b(t⃗)⟩
14:             execute all insertions delta-rules for b:
15:                 ⟨+, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
16:         if δ is a deletion update ⟨-, b(t⃗)⟩
17:             execute all deletion delta-rules for b:
18:                 ⟨-, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
19:         for all derived insertion (deletion) updates u do
20:             U^ν.insert(u)
21:         end for
22:     end while
23:     U ← U^ν.flush
24:     for all predicates p do
25:         P ← (P ⊎ I_p) \ D_p; I_p ← ∅; D_p ← ∅
26:     end for
27: end while
```

4.2.1 A Synchronous SN Algorithm

Algorithm 2 is a synchronous SN algorithm. There, all the updates in $\mathcal{U}$ (lines 2 – 10) are picked to fire delta-rules (lines 11–22) creating new updates, which are inserted in $\mathcal{U}$ (line 23), and then the changes are committed to the state (lines 24–26), where the operation *flush* in line 23 denotes that all the elements from $\mathcal{U}^\nu$ are moved to $\mathcal{U}$.

The main difference between Algorithm 1 and Algorithm 2 is that in Algorithm 2, all nodes are synchronized at the end of each iteration. In one iteration, all updates at the beginning of the iteration are processed by the corresponding nodes and updates created are sent accordingly. However, the updates that are created are not processed until the beginning of the next iteration. Nodes need to synchronize with one another so that no node is allowed to start the execution of the next iteration if there are some nodes that have not finished processing all the updates in its local queue in the current iteration or have not received all the updates generated by other nodes in the current iteration. On the other hand, Algorithm 1 allows each node to pick and process any one update available at the time of the pick.

For instance, if we apply SN to the same example discussed in Section 3.2, then all updates in $\mathcal{U}$:

$\mathcal{U}$ = {⟨+, link(d,f)⟩, ⟨+, link(a,f)⟩, ⟨-, link(a,b)⟩}

are necessarily picked and are used to fire delta-rules creating the following set of new updates:

$$\{\langle +, \texttt{hop(a,g)}\rangle, \langle +, \texttt{hop(d,g)}\rangle, \langle +, \texttt{hop(a,f)}\rangle, \langle -, \texttt{hop(a,c)}\rangle, \langle -, \texttt{hop(a,h)}\rangle\}$$

At the end of the while-loop, the updates picked are committed in the state. The facts `link(d,f)` and `link(a,f)` are inserted into the state, while the fact `link(a,b)` is deleted from it. The iteration repeats by using all the new updates created above.

Interestingly, the operational semantics for Algorithm 2 can also be defined in terms of the three basic commands: *pick*, *fire*, and *commit*. In particular an iteration of the outermost loop in Algorithm 2 corresponds exactly to an SN-iteration. Differently from PSN-iterations, where only a single update is picked at a time, SN-iterations are complete-iterations that pick *all* updates.

DEFINITION 6 (SN-iteration). *A complete-iteration is an* SN-iteration *if the multiset of updates used by the pick commands contains all updates in the initial configuration $\mathcal{U}$.*

DEFINITION 7 (SN execution). *We call a computation run an SN execution if it can be partitioned into a sequence of SN-iterations, and in the last configuration $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{E}$ are empty.*

4.2.2 Correctness Statement

In this section we prove that the Algorithm 2 is correct. For this we need to introduce the following set of definitions.

We keep track of the multiplicity of facts by distinguishing between different occurrences of the same fact in the following form: we label different occurrences of the same base fact with different natural numbers and label each occurrence of the same derived fact with the derivation supporting it. Consider, for example, the program from Section 2.2:

```
{p :- s,t,r;  s :- q;  s :- u;   t :- u;  q :-;  u :-}.
```

The state of the above program using multiset-semantics is actually interpreted in our proofs as the set of annotated facts:

$$\{\texttt{s}^{\Xi_1}, \texttt{s}^{\Xi_2}, \texttt{t}^{\Xi_3}, \texttt{q}^1, \texttt{u}^1\}$$

. The two occurrences of `s` are distinguished by using the derivations trees Ξ_1 and Ξ_2. The former is a derivation tree with a single leaf $\texttt{q}^1$ and the latter is a derivation tree with a single leaf $\texttt{u}^1$. We elide these annotations whenever they are clear from the context. These annotations are only used in our proofs as a formal artifact to distinguish different occurrences of facts.

We use the following notation throughout the rest of this section: given a multiset of updates $\mathcal{U}$, we write $\mathcal{U}^t$ to denote the multiset of facts in $\mathcal{U}$. Given a program $\mathcal{P}$, let V be the state of a program $\mathcal{P}$ given the set of base facts E, and let V^ν be the state of $\mathcal{P}$ given the set of facts $E \uplus I^t \setminus D^t$, where I and D are, respectively, a multiset of insertion and deletion updates of base facts. We assume that $D^t \subseteq E \uplus I^t$.

We write Δ to denote the multiset of insertion and deletion updates of facts such that V^ν is the same multiset resulting from applying the insertions and deletions in Δ to V. We write $\Delta[i]$ to denote the multiset of insertion and deletion updates of facts in Δ such that $\langle U, p(\vec{t})\rangle \in \Delta[i]$ if and only if $p(\vec{t})$ is supported by a derivation of height i. In an execution of Algorithm 2, we use $\mathcal{U}[i]$ to denote the multiset of updates at the beginning of the i^{th} iteration, and $\mathcal{U}[i,j]$ to denote the union of all multisets $\mathcal{U}[k]$ such that $i \leq k \leq j$.

Continue our example, the state of this program is the multiset of annotated facts $V = \{\texttt{s}^{\Xi_1}, \texttt{s}^{\Xi_2}, \texttt{t}^{\Xi_3}, \texttt{q}^1, \texttt{u}^1\}$. If we, for example, delete the base fact $\texttt{u}^1$, then the resulting state changes to $V^\nu = \{\texttt{s}^{\Xi_1}, \texttt{q}^1\}$, where the difference set is

$$\Delta = \{\langle -, \texttt{u}^1\rangle, \langle -, \texttt{s}^{\Xi_2}\rangle, \langle -, \texttt{t}^{\Xi_3}\rangle\},$$
$$\Delta[0] = \{\langle -, \texttt{u}^1\rangle\}, \text{ and } \Delta[1] = \{\langle -, \texttt{s}^{\Xi_2}\rangle, \langle -, \texttt{t}^{\Xi_3}\rangle\}.$$

Before proving the correctness of Algorithm 2, we formally define correctness, which is similar to the definition of *eventual consistency* used by Loo et al. [10] in defining the correctness of declarative networking protocols.

DEFINITION 8 (Correctness). *We say that an algorithm correctly maintains the state if it takes as input, a program $\mathcal{P}$, the state V based on base facts E, a multiset of insertion updates I and a multiset of deletion updates D, such that $D^t \subseteq E \uplus I^t$; and the resulting state when the algorithm finishes is the same as V^ν, which is the state of $\mathcal{P}$ given the set of facts $E \uplus I^t \setminus D^t$.*

In particular, we can prove that Algorithm 2 is indeed correct according to the definition above. It corresponds to maintenance algorithms that use semi-naïve strategies. The proofs which can be found in [14] are quite interesting. It is non-trivial to find the invariants needed for the proofs.

THEOREM 9 (Correctness of SN). *Given a non-recursive* DDlog *program $\mathcal{P}$, a multiset of base facts, E, a multiset of updates insertion updates I and deletion updates D to base facts, such that $D^t \subseteq E \uplus I^t$, Algorithm 2 correctly maintains the state of the program when it terminates.*

4.3 Relating SN and PSN executions

Our final goal is to prove the correctness of PSN. With the correctness result of Algorithm 2 in hand, now we are left to prove that Algorithm 1 computes the same result as Algorithm 2. At a high-level we would like to show that given any PSN execution, we can transform it into an SN execution without changing the final result of the execution. This transformation requires two operations: one is to permute two PSN-iterations so that a PSN execution can be transformed into one where the updates are picked in the same order as in an SN execution; the other is to merge several PSN-iterations into one SN-iteration. We need to show that both of these operations do not affect the final configuration of the execution.

Definitions. Let $s \xrightarrow{sn} (\mathcal{U})s'$ and $s \xrightarrow{psn} (\mathcal{U})s'$ denote, respectively, an execution from configuration s to s' using an SN iteration and a PSN iteration. We annotate the updates used in the iterations in the parenthesis after the arrow. We write $s \overset{a}{\Longrightarrow} s'$ to denote an execution from s to s' using multiple SN iterations, when a is *sn*; or PSN iterations, when a is *psn*. Let $s \Longrightarrow s'$ denote an execution from s to s' using multiple complete iterations. We write $\sigma_1 \rightsquigarrow \sigma_2$ if the existence of execution σ_1 implies the existence of execution σ_2. We write $\sigma_1 \leftrightsquigarrow \sigma_2$ when $\sigma_1 \rightsquigarrow \sigma_2$ and $\sigma_2 \rightsquigarrow \sigma_1$.

An update u is classified as *conflicting* if it is supported by a proof containing a base fact that was inserted (in I^t) and another fact that was deleted (in D^t). We say u and $\bar{u}$ are a pair of *complementary updates* if u is an insertion (deletion) of predicate p, and $\bar{u}$ is a deletion (insertion) of p. Intuitively, conflicting updates are temporary updates that appear in the execution of incremental maintenance algorithms but that do not affect the final configuration. The effect of a deletion update cancels the effect of the corresponding insertion update. Lemma 13 formalizes this intuition, and we will explain later in this section.

Permuting PSN-iterations. The following lemma states that permuting two PSN-iterations that are both insertion (deletion) updates leaves the final configuration unchanged. So in our example execution described in Section 3.2, it does not matter whether the update $\langle +, \texttt{link(a,f)}\rangle$ is picked before or after the update $\langle +, \texttt{link(d,f)}\rangle$. The set of updates after these two updates are

picked is the same, namely the set of updates: $\{\langle +, \texttt{hop(a,g)}\rangle, \langle +, \texttt{hop(a,f)}\rangle\}$.

LEMMA 10 (Permutation – same kind).
Given an initial configuration s,
$s \xrightarrow{psn} (\{\langle U, r_1\rangle\})s_1 \xrightarrow{psn} (\{\langle U, r_2\rangle\})s'$
$\leftrightsquigarrow$
$s \xrightarrow{psn} (\{\langle U, r_2\rangle\})s_2 \xrightarrow{psn} (\{\langle U, r_1\rangle\})s'$ *,where* $U \in \{+, -\}$.

The proof, given in [14], proceeds by considering all possible ways that an update can fire a rule and showing that the same set of updates are created when we permute the order in which the updates are picked.

However, permuting a PSN-iteration that picks a deletion update over a PSN-iteration that picks an insertion update might generate new updates. Consider a program consisting of the rule p :- r_1, r_2 and assume that r_2 is in the state. Furthermore, assume the updates $\{\langle +, r_1\rangle, \langle -, r_2\rangle\}$. If the deletion update is picked before the insertion update, no delta-rule is fired. However, if we pick the insertion rule first, then the rule above is fired twice, one propagating an insertion of p and the other propagating a deletion of p. However, the new updates are necessarily conflicting updates. This is formalized by the statement below. The side condition that $r_1 \neq r_2$ captures the semantics of the pick command in that deletion updates are only picked if the facts to be deleted are already in the state.

LEMMA 11 (Permutation – different kind).
Given and initial configuration s
$s \xrightarrow{psn} (\langle +, r_1\rangle)s_1 \xrightarrow{psn} (\langle -, r_2\rangle)\langle \mathcal{K}', \mathcal{U}' \uplus \Delta, \emptyset, \emptyset\rangle$
$\leftrightsquigarrow$
$s \xrightarrow{psn} (\langle -, r_2\rangle)s_2 \xrightarrow{psn} (\langle +, r_1\rangle)\langle \mathcal{K}', \mathcal{U}', \emptyset, \emptyset\rangle$,
where $r_1 \neq r_2$ *and* Δ *is a (possibly empty) multiset containing pairs of complementary conflicting updates.*

The proof is very similar to the proof of Lemma 10.

From PSN iterations to an SN iteration and back. The second operation we need for transforming a PSN execution into an SN execution is merging a PSN-iteration with a complete-iteration to form a bigger complete-iteration.

Similarly to the case when permuting PSN-iterations of different kinds, merging PSN iterations may change the set of conflicting updates. For example, consider a program consisting of a single rule p :- r, q, the initial state $\{q\}$, and the multiset of updates $\{\langle +, r\rangle, \langle -, q\rangle\}$. If both updates are picked in a complete-iteration, then an insertion update, $\langle +, p\rangle$, is created by firing the delta-rule $\langle +, p\rangle$:- $\Delta r, q$ using the insertion update $\langle +, r\rangle$. Similarly a deletion update $\langle -, p\rangle$ is created by firing the delta-rule $\langle -, p\rangle$:- $r^{\nu}, \Delta q$ and the deletion update $\langle -, q\rangle$. However, if we break the complete-iteration into two PSN-iterations, the first picking the deletion update and the second picking the insertion update, then no delta-rule is fired. We prove the following:

LEMMA 12 (Merging Iterations). *Let* $\mathcal{U}$ *be a multiset of updates such that the multiset* $\{u\} \uplus \mathcal{H} \subseteq \mathcal{U}$ *and let* $s = \langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle$ *be an initial configuration.*
$s \Longrightarrow (\{u\} \uplus \mathcal{H})\langle \mathcal{K}', \mathcal{U}' \uplus F_1, \emptyset, \emptyset\rangle$
$\leftrightsquigarrow$
$s \Longrightarrow (\mathcal{H})\langle \mathcal{K}_2, \mathcal{U}' \uplus \{u\} \uplus F_1', \emptyset, \emptyset\rangle \xrightarrow{psn} (u)\langle \mathcal{K}', \mathcal{U}' \uplus F_2, \emptyset, \emptyset\rangle$
Where F_1 *and* F_2 *only differ in pairs of complementary conflicting updates.*

Lemma 12 actually give us for free, the ability to break a complete SN-iteration into several PSN-iterations.

For example, we can use the lemma above to transform the SN-iteration shown in Section 4.2.1 where we pick all the updates appearing in the set of initial updates:

$$\{\langle +, \texttt{link(d,f)}\rangle, \langle +, \texttt{link(a,f)}\rangle, \langle -, \texttt{link(a,b)}\rangle\}$$

into a sequence of three PSN-iterations where these updates are picked one by one in any order. In this particular case, there are no conflicting updates created. The resulting sets of updates in both executions are the same:

$$\{\langle +, \texttt{hop(a,g)}\rangle, \langle +, \texttt{hop(d,g)}\rangle, \langle +, \texttt{hop(a,f)}\rangle, \langle -, \texttt{hop(a,c)}\rangle, \langle -, \texttt{hop(a,h)}\rangle\}.$$

Dealing with Conflicting Update Pairs. Next, we prove that conflicting updates do not interfere with the final configuration when using PSN executions. Intuitively, we will rely on the following observations: (1) All updates generated by firing delta-rules for conflicting updates are also conflicting updates. (2) A pair of complementary conflicting updates generate pairs of complement conflicting updates. For example, consider adding the rule v :- p to the example given before Lemma 12. Then the conflicting update $\langle +, p\rangle$ would propagate the update $\langle +, v\rangle$. The latter update is also conflicting because the fact p is supported by a fact q which is to be deleted. Moreover, when the deletion of q "catches up," then the complementary update $\langle -, v\rangle$ is created and cancels the effect of the conflicting update $\langle +, v\rangle$. Consequently, a PSN execution that contains a pair of complementary conflicting updates in its initial configuration can be transformed into another PSN execution that does not contain these updates and that the final configurations of the two executions are the same. The following lemma precisely states that.

LEMMA 13. *Let* $\Delta = \{\langle +, p\rangle, \langle -, p\rangle\}$ *be a multiset containing a pair of complementary conflicting updates, then*
$\langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle \overset{psn}{\Longrightarrow} s \quad \leftrightsquigarrow \quad \langle \mathcal{K}, \mathcal{U} \uplus \Delta, \emptyset, \emptyset\rangle \overset{psn}{\Longrightarrow} s.$

Its proof relies on the termination arguments for PSN algorithm for non-recursive programs. For recursive programs, it is possible that a pair of complementary conflicting updates will generate infinite number of complementary conflicting updates; and therefore the transformation process may never terminate.

Correctness of Basic PSN. Finally, using the operations above we can prove the following theorem, which establishes that PSN is sound and complete with respect to SN.

THEOREM 14 (Correctness of PSN w.r.t. SN). *Let* $s = \langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle$ *be an initial configuration. Then for non-recursive programs:*
$s \overset{psn}{\Longrightarrow} \langle \mathcal{K}, \emptyset, \emptyset, \emptyset\rangle \leftrightsquigarrow s \overset{sn}{\Longrightarrow} \langle \mathcal{K}, \emptyset, \emptyset, \emptyset\rangle$.

The above theorem states that the same derived facts that are created by SN are also created by PSN and vice-versa. The proof idea is that we can use the operations described in Lemmas 10, 11, and 12 to transform a PSN execution into an SN one and vice-versa. In particular, we use Lemmas 10 and 11 to permute PSN iterations so that updates are picked in the same order as an SN execution. Then we use Lemma 12 to merge PSN-iterations into SN-iterations. The conflicting updates that are created in the process of using such transformations are handled by Lemma 13. Hence, from Theorem 9, PSN is correct.

COROLLARY 15 (Correctness of basic PSN). *Given a non-recursive DDlog program* $\mathcal{P}$*, a multiset of base facts,* E*, a multiset of updates insertion updates* I *and deletion updates* D *to base facts, such that* $D^t \subseteq E \uplus I^t$*, then Algorithm 1 correctly maintains the state of the program.*

Discussion The framework of using three basic commands: *pick*, *fire*, and *commit* to describe PSN and SN algorithms can be used for specifying and proving formal properties about other SN-like algorithms. For instance, one can easily generalize the proof above to prove the correctness of algorithms where nodes pick

- $pick^1_I(\mathcal{S}, \langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K} \uplus \{(p^\nu(\vec{t}), \mathcal{S}, \mathcal{H}')\}, \mathcal{U} \setminus \{\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H}')\rangle\}, \mathcal{E}\rangle$,
  provided $\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U}$ and $p(\vec{t}) \in \mathcal{S}$, where $\mathcal{H}' = \mathcal{H} \cup \{p(\vec{t})\}$.
- $pick^2_I(\mathcal{S}, \langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K} \uplus \{(p^\nu(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U} \setminus \{\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{E}\rangle$,
  provided $\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U}$ and $p(\vec{t}) \notin \mathcal{S}$.
- $pick^1_D(\mathcal{S}, \langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K} \setminus \{(p^\nu(\vec{t}), \mathcal{S}, \mathcal{H}')\}, \mathcal{U} \setminus \{\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H}')\rangle\}, \mathcal{E}\rangle$,
  provided $\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U}$ and $p(\vec{t}) \in \mathcal{S}$, where $\mathcal{H}' = \mathcal{H} \cup \{p(\vec{t})\}$.
- $pick^2_D(\mathcal{S}, \langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K} \setminus \{(p^\nu(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U} \setminus \{\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{E}\rangle$,
  provided $\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U}$ and $p(\vec{t}) \notin \mathcal{S}$.
- $fire(\mathcal{S}, u) = \langle \mathcal{K} \uplus \{(p(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}\rangle$, provided $u \in \mathcal{P}$, where $\mathcal{F} = firRules(u, \mathcal{K}, \mathcal{R})$.
- $commit_I(\mathcal{S}, \langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K}, \mathcal{U} \uplus \mathcal{F}, \mathcal{P} \setminus \{u\}, \mathcal{E} \uplus \{u\}\rangle$, provided $\langle +, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{E}$.
- $commit_D(\mathcal{S}, \langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle) = \langle \mathcal{K} \setminus \{(p(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}\rangle$, provided $\langle -, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{E}$.

Figure 3. Definitions for the basic commands that detect cycles. Here $\mathcal{S}$ is the configuration $\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle$.

multiple updates per iteration instead of just one update, as in PSN-iterations; or the complete multiset of updates available, as in SN-iteration. That is, we can transform an execution with arbitrary complete iterations into an SN execution and vice-versa. One first breaks the complete-iterations into PSN-iterations, obtaining a PSN execution. Then the proof follows in exactly the same way as before. This means that when implementing such systems, a node can pick all applicable updates that are in its buffer and process them in one single iteration, instead of picking them one by one, and the resulting algorithm is still correct.

5. Extended PSN Algorithm for Recursive Programs

Algorithm 1 and 2 use multiset-semantics. As a consequence, termination is not guaranteed when they are used to maintain states of recursive programs. Consider the following recursive program.

```
p(@1) :- a(@0)    q(@2) :- p(@1)    p(@1) :- q(@2)
```

Notice that `p` and `q` form a cycle in the dependency graph. Any insertion of the fact `p(@1)` will trigger an insertion of `q(@2)` and vice versa. Given an insertion of the fact `a(@0)`, neither Algorithm 1 nor Algorithm 2 terminate because the propagation of insertion updates of `q(@2)` and `p(@1)` do not terminate. Recursively defined predicates could have infinite number of derivations because of cycles in the dependency graph. In other words, in the multiset-semantics, such facts have infinite count. Neither Algorithm 1 nor Algorithm 2 have the ability to detect cycles.

One way to detect such cycles in a centralized setting is proposed in [12]. The main idea is to remember for any fact p, the set of facts, $\mathcal{S}$, called *derivation set*, that contains all the facts that are used to derive p. While maintaining the state, the algorithm checks whether a newly derived fact p appears in the set of facts supporting it. If this is the case, then there is a cycle, and p has infinite count. Whenever a fact with infinite count is detected, we store it in a second set, $\mathcal{H}$, called *infinite count set*. Future updates of p are not propagated to avoid non-termination.[2]

The same idea is applicable to the distributed setting. We formalize this by attaching the derivation and infinite count sets, $\mathcal{S}$ and $\mathcal{H}$, to facts both in states and updates. An annotated fact is of the form $(p, \mathcal{S}, \mathcal{H})$, where p is a fact, $\mathcal{S}$ is the derivation set of p, containing all the facts used to derive p, and $\mathcal{H}$ is a subset of $\mathcal{S}$ containing all the recursive facts that belong to a cycle in the derivation and therefore cause p to have an infinite count. In the example above, the state of facts without ν of the nodes would be:

$$\{(\texttt{a}, \emptyset, \emptyset), (\texttt{p}, \{\texttt{a}\}, \emptyset), (\texttt{q}, \{\texttt{p,a}\}, \emptyset), (\texttt{p}, \{\texttt{a,p,q}\}, \{\texttt{p}\}), \ldots\}$$

where we elide the `(@X)` symbols. The fact `p` in `(p,{a,p,q},{p})`, also appears in the set supporting it. This means that `p` appears in a cyclic derivation, and therefore p is in the set $\mathcal{H}$.

In order to maintain correctly the state, we adapt the definition of the basic commands accordingly. A summary of the rules are shown in Figure 3. Each *pick* rule in Figure 2 is divided into two rules. Once an update $u = \langle U, (p, \mathcal{S}, \mathcal{H})\rangle$ is picked from the multiset of updates by using either the transition rule $pick_I$ or $pick_D$, the algorithm first checks whether the fact is supported by a derivation tree that has a cycle (if $p \in \mathcal{S}$). If so, then p is added to the set $\mathcal{H}$; otherwise $\mathcal{H}$ remain unchanged. Notice that the updated state of p in $\mathcal{K}$ uses the updated $\mathcal{H}$ set. The *commit* rule is the same as before, except for the new presentation of facts.

The major changes in the operational semantics are in the *fire* rule, where the derivation set and the infinite count set need to be computed, when a delta-rule is fired and the propagation of updates to facts with infinite count need to be avoided. Given an update $\langle U, (b_i, \mathcal{S}_i, \mathcal{H}_i)\rangle$, in addition to computing all updates that are propagated from this update, the algorithm also constructs the corresponding derivation and infinite count sets, $\mathcal{S}$ and $\mathcal{H}$ as follows. Assume that the update $\langle U, p\rangle$ is propagated using a delta-rule with body $b^\nu_1, \ldots, b^\nu_i, \Delta b_i, b_{i+1}, \ldots, b_n$ and the facts $(b_j, \mathcal{S}_j, \mathcal{H}_j)$ where $1 \leq j \leq n$, then the derivation set for p is $\mathcal{S}_p = \{b_1, \ldots, b_n\} \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n$ and the infinite count set $\mathcal{H}_p = \mathcal{H}_1 \cup \cdots \cup \mathcal{H}_n$. In order to avoid divergence, we also need to make sure that an update of a fact with infinite count is not re-send. To do so, the algorithm only adds the update $\langle U, (p, \mathcal{S}_p, \mathcal{H}_p)\rangle$ to the multiset of updates $\mathcal{U}$, if it is not part of cycle that has been already computed ($p \notin \mathcal{H}_p$).

Returning to the previous example, when the update inserting the fact `p(@1)` arrives for the second time at node `1`, this update would contain the derivation set $\mathcal{S}$ = `{a(@0),p(@1),q(@2)}`. Since the fact `p(@1)`$\in \mathcal{S}$, node `1` detects the cycle in the derivation and adds the fact `p(@1)` to the infinite count set $\mathcal{H}$. As `q(@2)` is not in $\mathcal{H}$, the insertion update of `q(@2)` is sent to node `2`. However, when this update is processed, creating a new insertion of `p(@1)`, this new insertion is not sent back to `1` because `p(@1)` is in the infinite count set, which means that it is part of a cycle that has already been computed. Therefore, computation terminates. In fact, the derivation set and infinite count set guarantee termination of PSN on any recursive *DDlog* program.

THEOREM 16 (Finiteness of PSN that detects cycles). *Let $\mathcal{S}$ be an initial configuration and $\mathcal{R}$ be a* DDlog *program. Then all PSN executions using $\mathcal{R}$ and from $\mathcal{S}$ have finite length.*

[2] Notice that the derivation set of a fact is not the same as the annotation used before in our proofs to distinguish different occurrences of the same fact. The former is part of the algorithm, while the latter is only used in our proofs.

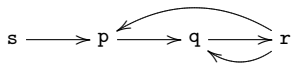

Figure 4. Dependency graph of a propositional program.

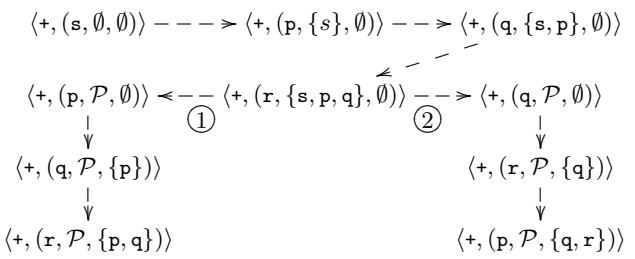

Figure 5. Sequence of updates created in an execution of PSN that detect cycles when inserting the base fact s. Here $\mathcal{P} = \{s, p, q, r\}$.

The proof of the theorem relies on the fact that while executing PSN that detects cycles, the size of the derivation set, $\mathcal{S}$ and the infinite set, $\mathcal{H}$, of updates increase. Since there are finitely many different facts in a program, there is an upper bound on the sizes of these sets. Hence, there is a global bound on the number of possible updates created in a run and therefore PSN that detects cycles terminates.

COROLLARY 17. *The PSN algorithm that detects cycles always terminates.*

Consider the following program with five clauses:

```
p :- s; q :- p; r :- q; p :- r; q :- r,
```

whose dependency graph is depicted in Figure 4 and contains multiple dependency cycles. Figure 5 contains the sequence of updates created when executing PSN that detects cycles starting from an update inserting the base fact s. The branches 1 and 2 are created when $\langle +, (r, \{s, p, q\}, \emptyset)\rangle$ is used to fire delta-rules. At the end of these two branches, no more updates are created. At the end of branch 1, processing the update $\langle +, (r, \mathcal{P}, \{p, q\})\rangle$ does not propagate any updates, since it could only propagate an insertion of q and of p. However, both q and p are in its infinite set, which means that they have infinite count, and therefore such updates are not created. Similarly, in the branch 2, processing the update $\langle +, (p, \mathcal{P}, \{q, r\})\rangle$ does not propagate new updates, since q is in its infinite count set. In the branches 1 and 2, the algorithm detects that all facts in $\{p, q, r\}$ have an infinite count. For instance, the first PSN-iteration in branch 1, which processes the update $\langle +, (p, \mathcal{P}, \emptyset)\rangle$, consists of the basic commands $pick_I^1$, *fire*, and $commit_I$. In the $pick_I^1$ the fact p is added to the infinite set, $\emptyset$, because p appears in the supporting set, $\mathcal{P}$. Hence, at the end of this iteration, by the $commit_I$ command, the fact $(p, \mathcal{P}, \{p\})$ is added to the state, which indicates that p has infinite count since p is in the infinite count set of this fact.

As we discuss in the companion tech report [14], the use of the annotated facts does not change the correspondence between PSN executions and SN executions. Once we show that PSN that detects cycles terminates, the same transformations used in Section 4 can be used to show that a PSN execution can be transformed into an SN execution and vice-versa, showing hence that PSN that detects cycles is correct.

COROLLARY 18 (Correctness of PSN). *Given any* DDlog *program $\mathcal{P}$, a multiset of base facts, E, a multiset of updates insertion updates I and deletion updates D to base facts, such that $D^t \subseteq E \uplus I^t$, then the PSN algorithm that detects cycles correctly maintains the state of the program.*

6. Comparison with Existing Incremental Maintenance Algorithms

We compare our algorithm with existing incremental maintenance algorithms. We discuss limitations of these existing approaches and how our algorithms improve them.

Delete and Re-derive. Gupta *et al.* proposed an algorithm in their seminal paper [7] on incrementally maintaining logic programs in a centralized setting, called DRed (Delete and Re-derive). DRed [7] maintains a state by using set-semantics. DRed does not keep track of the number of supporting derivations for any fact. Whenever a fact, p, is deleted, DRed eagerly deletes all the facts that are supported by a derivation that contains p. Since some of the deleted facts may be supported by alternative derivations that do not use p, DRed re-derives them in order to maintain a correct state.

Re-deriving facts in a distributed setting is expensive due to high communication overhead, as demonstrated in [9]. Consider, for example, the topology depicted in Figure 1, taken from [7]. There are two ways to reach the node `c` from the node `a`, one passing the node `b` and the other through the node `d`. Therefore the fact `reachable(@a,c)` is supported by two derivations. However, when using set-semantics, DRed only stores one copy of `reachable(@a,c)` at the node `a`. Assume that at some point the link from node `a` to the node `b` is broken, that is, the fact `link(@a,b)` is deleted. Then in DRed's deletion phase, the deletion of this fact propagates the deletion of `reachable(@a,b)`, which similarly will propagate the deletion of `reachable(@a,c)` and of `reachable(@a,h)`. Then DRed's re-derive phase starts, which checks which facts that were deleted in the deletion phase can be re-derived using an alternative derivation. In this case, all the deleted facts (`reachable(@a,b)`, `reachable(@a,c)`, and `reachable(@a,h)`) are re-derivable using other derivations. All the `reachable` facts derived using the path from `a` to `b` that passes through `d` have to be sent cross the network. For example `reachable(@d,c)` is send to `a` in order to re-derive the fact `reachable(@a,c)`.

Our algorithm (Algorithm 1) uses multiset-semantics to keep track of the number of supporting derivations of any fact. So, whenever a fact is deleted, such algorithm just needs to reduce its multiplicity by one, and whenever its multiplicity is zero, the fact is deleted from the state. Algorithm 1 incurs less communication than DRed. Our extended algorithm (Section 5) annotates each predicate with the set of supporting facts. Compared with DRed, this algorithm incurs higher communication overhead in a workload where there are no deletions. In the presence of deletions, our algorithm results in lower communication overhead, since the deletion of a fact does not require the construction of alternative derivations.

Original PSN algorithm. The original PSN algorithm was proposed by Loo *et al.*[10]. Our paper extends the original proposal in several ways. First, Loo *et al.* consider only linear recursive terminating Datalog programs. We consider the complete Datalog language including non-linear recursive programs. Second, we relax the assumptions in the original proposal: instead of assuming that the transmission channels are FIFO, which is unrealistic in many domains, we do not make any assumption about the order in which updates are processed. In other words, we do not assume the existence of a coordinator in the system. An important improvement is that the PSN algorithm proposed in this paper is proven to terminate and maintain states correctly. As pointed out in our previous work [13], the PSN algorithm as presented in [10] may produce unsound results and the use of the count algorithm [7] leads to non-termination. We elaborate further on the former problem of the original PSN algorithm.

The original PSN performs the following operation: whenever an update reaches a node, the update is not only stored at the end of the node's update queue, but also immediately used to update the

Node 1 :	$\{\}[]$	Burst	$\{\}[]$			$\{p\}[\langle +, p\rangle]$	Dequeue	$\{p\}[\langle +, p\rangle]$	Dequeue	$\{p\}[]$
Node 2 :	$\{s, t\}[]$	of	$\{r, s, t\}[\langle +, r\rangle]$	Dequeue		$\{r, s, t\}[]$	$\langle -, q\rangle$	$\{r\}[\langle -, s\rangle, \langle -, t\rangle]$	all	$\{r\}[]$
Node 3 :	$\{q\}[]$	updates.	$\{\}[\langle -, q\rangle]$	$\langle +, r\rangle$		$\{\}[\langle -, q\rangle]$	$\langle -, u\rangle$	$\{\}[]$	updates	$\{\}[]$
Node 4 :	$\{u\}[]$	$\longrightarrow$	$\{\}[\langle -, u\rangle]$	$\longrightarrow$		$\{\}[\langle -, u\rangle]$	$\longrightarrow$	$\{\}[]$	$\longrightarrow^*$	$\{\}[]$

Figure 6. PSN computation-run resulting in an incorrect final state. The i^{th} row depicts the evolution of the state, in curly-brackets, and the update queue, in brackets, of node i. The updates in the arrows are the ones dequeued by PSN and used to update the state of the nodes. We also elide the `(@X)` in facts.

node's local state: the fact in the update is immediately inserted into or deleted from the node's state. This procedure, however, leads to unsound results if channels are not FIFO. Consider the following *DDlog* program, which is the same program as shown in Section 2.2, but now distributed over four nodes. The global state of this program is {`s(@2), t(@2), q(@3), u(@4)`}:

```
node2: p(@1) :- s(@2), t(@2), r(@2).
node3: s(@2) :- q(@3).
       q(@3) :-.
node4: t(@2) :- u(@4).
       u(@4) :-
```

Consider the PSN computation-run depicted in Figure 6 (based on the original algorithm). At the first transition, there is a burst of updates inserting the base fact r and deleting the base facts q and u, where we elide the (@X) symbols. When these updates are created, they are not only stored in the nodes' queues but also used to update the state of the nodes (first transition in Figure 6). Then when the update $\langle$`+`$, r\rangle$ is dequeued and processed, a new update inserting p is created (second transition in Figure 6). When the updates $\langle$`-`$, q\rangle$ and $\langle$`-`$, u\rangle$ are processed, they create the updates $\langle$`-`$, s\rangle$ and $\langle$`-`$, t\rangle$ (third transition in Figure 6). In the final transitions, none of the updates deleting `s` or `t` trigger the deletion of `p` because `t` and `u` are no longer in node 2's state and the bodies of the respective deletion rules are not satisfied. Hence, the predicate `p` is entailed after the original PSN terminates although it is not supported by any derivation.

Our algorithms correct this error by delaying updates to the facts until after updates are processed.

PSN with annotated facts. After the original PSN algorithm, Liu *et al.* proposed in [9] a new PSN algorithm where facts are annotated in order to handle the known problem that the original PSN does not terminate. Differently from our approach, Liu *et al.* only track the base facts used in the derivation, while our *derivation set* contains all facts (including intermediate derived facts) used for each derivation. Moreover, as with the original PSN algorithm, Liu *et al.* also assume the existence of coordinator in the system enforcing that all transmission channels are FIFO. Under this assumption, Liu *et al.* show that their PSN algorithm terminates.

However, by using only base facts, it is not possible, without assuming that the transmission channels used are FIFO, to differentiate an update that is the result of computing a cyclic derivation from an update that arrived out-of-order. When messages are processed out of order, the algorithm proposed in [9] yields unsound results, illustrated below.

Consider the following program also used in Section 5 that contains cycles and for which original PSN does not terminate:

```
a(@0) :-; p(@1) :- a(@0); q(@2) :- p(@1); p(@1) :- q(@2)
```

In [9], the state of this program is represented as the set {`(a,{a}), (p,{a}), (q,{a})`} where we elide the $(@X)$ symbols. All facts are derived by only using the base fact `a` and therefore their annotations consist only of the base fact `a`. An update inserting `(p,{a})` could be derived due to a derivation with no cycles or due to a cyclic derivation obtained by using the last two rules of the program. In order to avoid divergence, the latter type of updates resulting from cyclic derivations need to be discarded. Assume that there is a deletion of `a`, represented by a deletion update $\langle$`-, (a,{a})`$\rangle$. When this update is processed, node `1` creates $\langle$`-, (p,{a})`$\rangle$, which is processed by node `2`, creating the update $\langle$`-, (q,{a})`$\rangle$. Finally, node `2` processes the latter, creating again the deletion update $\langle$`-, (p,{a})`$\rangle$. When this update is received by node `1`, the fact `(p,{a})` is not in the state, as it was deleted by the first deletion update. Therefore, node `1` can safely conclude, under the assumption of FIFO channels, that the latter update is due to a cyclic derivation. Hence it just discards it and the algorithm terminates.

It is easy to show that discarding eagerly such deletion updates yields unsound results when one relaxes the assumption of FIFO channels. Consider the same program above, but two conflicting updates: $\langle$`-, (a,{a})`$\rangle$ and $\langle$`+, (a,{a})`$\rangle$. If the deletion update is processed first by node `0`, it will be discarded since the fact `(a,{a})` is not present in its state. The insertion update on the other hand would be processed, generating eventually new insertion updates for all the facts in the program. Hence, the final state obtained by their algorithm is `(a,{a}), (p,{a}), (q,{a})`, whereas the correct state is the empty set.

Our algorithm annotates each predicate with all the predicates used to derive it, which include not only the base predicates, but also intermediate predicates. We have shown in Section 5 that we can detect cycles properly, even in the presence of message re-ordering. Finally, Liu *et al.*'s algorithm is only experimentally evaluated but not formally proven correct.

7. Additional Related Work

In contrast to our approach, MELD [5] simply attaches to each fact the height of the supporting derivation. Although they are able to perform many optimizations with such type of annotations, simply attaching the height of derivations to facts is not enough to detect cycles in derivations and therefore it is not enough to avoid divergence by itself. They address this problem by synchronizing nodes and not allowing nodes to compute until they receive the response from other nodes that all the deletions propagated from a deletion of a base fact have been processed. As expected, performance can be greatly affected since an unbounded number of nodes might need to be synchronized at the same time due to cascading derivations. We believe that their work can directly leverage the results in this paper.

In an attempt to generalize Loo *et al.*'s work [10], Dedalus [4] relaxes the set of assumptions above by no longer assuming that messages always reach their destination. The main difficulty when considering message loss is that the semantics does not relate well with the semantics in the Datalog literature. Depending on whether a message is lost or not, the final states computed by their evaluation algorithms can be considerably different. Therefore, it is not clear what is the notion of correctness in such systems. We believe that probabilistic models where messages are lost with certain probability can be used, and we leave this for future work.

In the agent programming community, several languages that allow for the update of knowledge bases have been proposed. For

instance, [3] proposes a logic programming language that allows updates not only to base facts, but also to rules themselves. Differently from this paper, however, their work considers only a centralized setting. Moreover, a central difference from our work is that while [3] is concerned in extending logic programming languages so that programmers can specify updates, here we focus on algorithms that efficiently maintain states of distributed Datalog programs. An interesting direction for future work would be extend our results to also allow rule updates in a distributed setting.

Adjiman *et al.* in [2] use classical propositional logic to specify knowledge bases of agents in a peer-to-peer setting. They prove correct a distributed algorithm that computes the consequences of inserting a literal, that is, an atom or its negation, to a node (or peer). Since they use resolution in their algorithm, they are able to deduce not only the atomic formulas that are derivable when an insertion is made, but propositional formulas in general. While they are mainly interested in finding the resulting state from inserting a formula, we are interested in efficiently maintaining a state was previously computed. It is not clear how their approach can be used to update the consequences when a sequence of insertions and deletions are made to the knowledge base.

8. Conclusions and Future Work

Besides the correctness of the algorithm itself, our ultimate goal is to prove interesting properties about programs written in distributed Datalog. The correctness results in this paper allow us to first formally verify high-level properties of programs prior to actual deployment by relying on the well established semantics for centralized Datalog, then the verified properties carry over to the distributed deployment, because semantics for Distributed Datalog and centralized Datalog coincide.

In particular, we are interested in formal verification of implementations of networking protocols prior to actual deployment in declarative network setting [19, 20]. In order to do so, we need to extend this work to include additional language features present in declarative networking including function symbols and aggregates. Since Datalog programs with arbitrary functions symbols may not terminate, we are investigating if we can extend existing analysis techniques [8] developed for centralized Datalog with function symbols to determine when *DDlog* programs with function symbols terminate. It turns out that it is not an easy task to develop efficient and correct algorithms that maintain logic programs incrementally in the presence of aggregate functions. We are looking into adapting existing work, such as [17] in incremental view maintenance in a centralized setting to fit our needs.

Acknowledgements We would like to thank Iliano Cervesato, Dale Miller, Juan Antonio Navarro Pérez, Frank Pfenning Andrey Rybalchenko, Val Tannen, and Anduo Wang for helpful discussions.

This material is based upon work supported by the MURI program under AFOSR Grant No: FA9550-08-1-0352 and by the NSF Grants IIS-0812270 and CNS-0845552. Additional support for Scedrov and Nigam from ONR Grant N00014-07-1-1039 and from NSF Grants CNS-0524059 and CNS-0830949. Nigam was also supported by the Alexander von Humboldt Foundation. Scedrov was also partially supported by ONR grant N000141110555.

References

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

[2] P. Adjiman, P. Chatalic, F. Goasdoué, M.-C. Rousset, and L. Simon. Distributed reasoning in a peer-to-peer setting: application to the semantic web. *J. Artif. Int. Res.*, 25(1):269–314, 2006.

[3] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming. In *KR*, pages 98–111, 1998.

[4] P. Alvaro, W. 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.

[5] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, October 2007.

[6] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, January 2010.

[7] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 1993.

[8] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A framework for testing safety and effective computability. *J. Comput. Syst. Sci.*, 52(1):100–124, 1996.

[9] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE)*, March 2009.

[10] 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 *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2006.

[11] N. P. Lopes, J. A. Navarro, A. Rybalchenko, and A. Singh. Applying Prolog to develop distributed systems. In *Proceedings of the Twenty-sixth International Conference on Logic Programming (ICLP)*, July 2010.

[12] I. S. Mumick and O. Shmueli. Finiteness properties of database queries. In *Australian Database Conference*, pages 274–288, 1993.

[13] V. Nigam, L. Jia, A. Wang, B. T. Loo, and A. Scedrov. An operational semantics for network datalog. In *Third International Workshop on Logics, Agents, and Mobility (LAM)*, July 2010.

[14] V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining Distributed Recursive Views Incrementally. Technical Report No. MS-CIS-11-06, UPENN, 2011.

[15] V. Paxson. End-to-end routing behavior in the internet. In *SIGCOMM*, August 1996.

[16] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[17] R. Ramakrishnan, K. A. Ross, D. Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *Proceedings of the 1994 International Symposium on Logic programming (SLP)*, 1994.

[18] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2):233–277, 1996.

[19] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, Jan. 2009.

[20] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu. Formally Verifiable Networking. In *SIGCOMM HotNets-VIII*, 2009.