Maintaining Distributed Recursive Views Incrementally

Vivek Nigam
UPENN, USA
vnigam@math.upenn.edu

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

This paper proposes an algorithm to compute incrementally the changes to distributed recursive database views in response to insertions and deletions of base facts. Our algorithm uses a pipelined semi-naïve (PSN) evaluation strategy introduced in declarative networking. Unlike prior work, our algorithm is formally proven to be correct for recursive query computation in the presence of message reordering in the system. Our proof proceeds in two stages. First, we show that all the operations performed by our PSN algorithm computes the same set of results as traditional centralized semi-naïve evaluation. Second, we prove that our algorithm terminates, even in the presence of cyclic derivations due to recursion.

1. INTRODUCTION

One of the most exciting developments in computer science in the past years is the fact 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 be performed by different machines as well, *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 declarative programming languages, based on Prolog and Datalog, that allow one to write programs for distributed systems such as networks and multi-agent robotic systems, *e.g.* Network Datalog (*NDlog*) [11], MELD [4], Netlog [7], DAHL [12], Dedalus [3]. When programming in these declarative languages, programmers usually do not need to specify *how* computation is done, but rather *what* is to be computed; and therefore, declarative programs tend to be more readable, portable, and orders of magnitude smaller than usual imperative implementations.

One of the key observations that these languages rely on is that distributed systems, such as networking and multi-agent robotic systems, deal at their core with maintaining distributed states by allowing each node to compute locally and then propagate its local states to other nodes (agents) 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 table to its neighbors. We can think of these systems as distributed database views, where not only base facts are distributed, but the rules are also distributed among different nodes in the network. Computation in these systems can be regarded as computing distributed database views.

Similarly to its centralized counterpart, one of the main challenges of implementing these distributed views is how to efficiently and correctly update them when the base facts change. For instance, in the network setting, when a new link in the network has been established or an old link has been broken, one needs to update the routing tables to reflect the changes in the base predicates. It is often 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, including those that have been previously propagated. A much better approach is to maintain distributed databases incrementally. That is, instead of reconstructing the entire database, one only modifies previously derived tuples that are affected by the changes of the base tuples, while the remaining facts are left untouched.

Gupta *et al.* proposed an algorithm in their seminal paper [8] on incrementally maintaining database views in a centralized setting, called DRed (Delete and Rederive). DRed first deletes all tuples that might be affected by the changes to base tuples, then rederives tuples that are still derivable. As shown in [10], it turns out that DRed is not practical in a distributed setting, due to high communication overhead incurred by rederivations of tuples.

This paper develops techniques for incrementally maintaining recursive views in a distributed setting, while avoiding as much as possible synchronization among agents (nodes). That is, no agent is supposed to *stop* computing because some other agent has not concluded its computation and we do not assume the existence of any coordinator in the system. Synchronization requires extra communication between nodes, which comes at a huge performance penalty. In particular, we propose an asynchronous incremental view maintenance algorithm, based on the *pipelined semi-naïve* (PSN) evaluation strategy proposed by Loo *et al.* [11]. PSN relaxes the traditional semi-naïve (SN) evaluation strategy 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's 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.

Most importantly, we formally prove the correctness for our PSN algorithm, which is lacking from existing work on distributed view maintenance. What makes the problem hard is the fact that we need to show that, in a distributed, asynchronous setting, the view computed by our algorithm is correct *regardless of the order in which updates are processed*. Unlike prior PSN proposals [11, 10], our PSN algorithm does not require that message channels are FIFO, which is for many distributed systems an unrealistic assumption.

Furthermore, guaranteeing termination for distributed recursive views also turns out to be tricky. In a centralized, synchronous setting termination is usually guaranteed by discarding new derivations of previously derived tuples by using evaluation mechanisms that use set-semantics, such as DRed. The use of set-semantics, however, requires nodes to re-derive tuples that are deleted, affecting performance in a distributed setting. When using multiset semantics, on the other hand, one does not need to re-derive tuples, since nodes keep track of all possible derivations for any tuple. However, one can no longer easily guarantee termination, as tuples might be supported by infinitely many derivations. Inspired by [13], we add to tuples annotations containing the set of facts (base and intermediate) that were used to derive it. We use this annotation to detect cycles in derivations, which is enough to detect when tuples are supported by infinitely many derivations, and hence guarantee termination.

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

- We propose a new PSN-algorithm to maintain views incrementally in a distributed setting (Section 3). This algorithm only deals with distributed non-recursive views.
- We formally prove that PSN is correct (Section 4). Instead of directly proving PSN maintains views correctly, we construct our proofs in two steps. First, we define a synchronous algorithm based on SN evaluation, 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 tuple with information about its derivation to ensure the termination of maintaining views for recursive programs (Section 5), and prove its correctness.

Finally, we discuss related work in Section 6, and conclude with some final remarks in Section 7.

2. PRELIMINARIES

In this section, we review the basic definitions of Datalog and introduce the language *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* [11], MELD [4], Netlog [7], and Dedalus [3].

2.1 Background: Datalog

A *Datalog* program consists of a (finite) set of logic rules and a query. The query is a ground fact, that is, , that is, a fact containing no variable symbols. A rule has 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; $h(\vec{t})$ is an atomic predicate called the head of the rule; $b_1(\vec{t_1}), \ldots, b_n(\vec{t_n})$ is a sequence of atomic predicates called the body; and the $\vec{t}$s are vectors of variables and ground terms. Semantically, the order of the elements in the body does not matter, but it does have an impact on how programs are evaluated (usually from left to right).

Any free variable in a Datalog rule is assumed to be universal quantified. Moreover, we assume that all variables appearing in the head of a rule also appear somewhere in its body. Following [17], we assume a *finite* signature of predicate and constant symbols, but no function symbols.

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, and the remaining predicates as derived predicates. Consider the following non-recursive Datalog 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, called *view* or state, is `{s, t, q, u}`. For this example, each predicate is supported by only one derivation and therefore the same view is obtained when using as intended semantics set semantics or multiset semantics. If we added, however, the clause `s :- u` to this program, then the view when using 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 another by using `u`. The view using set semantics can, however, be easily derived from the view obtained when using multiset semantics by simply eliminating the multiplicity of the tuples in the view. Therefore, since the algorithms proposed in this paper keep track of the multiplicity of tuples, they can be used for either multiset or set-semantics. We discuss at the end of this section, why we opt to use multiset-semantics.

2.2 Distributed Datalog

Location Specifiers.

To allow distributed computation, *DDlog* extends Datalog by augmenting its syntax with the location operator `@` [11], which specifies the location of a tuple. For instance, consider the following *DDlog* program, which calculates the reachability relation among nodes:

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

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

Rules `r1`–`r2` recursively derive `reachable(@S,D)` tuples, where each tuple represents the fact 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)`, while rule `r2` computes transitive reachability as follows: if there exists a link from `S` to `Z`, and `Z` knows that 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 tuple that are relevant to its own state. For example, the tuple `reachable(@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 tuples may need to be sent to the corresponding nodes as specified by the `@` operator.

Rule localization.

As illustrated by the rule `r2`, the predicates in the body of clauses can have different location specifiers indicating that they are stored on different nodes. To apply such a rule, predicates may need to be gathered from several nodes, possible different from where the rule resides. To have a clear defined semantics of the program, we apply

rule localization rewrite procedure as shown in [11] to make such communication explicit. The *rule localization* rewrite procedure transforms a program into an equivalent one 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 predicates. This procedure improves performance by eliminating the need of unnecessary communication among nodes, as a node only needs the tuples locally stored to derive a new fact. For example, the rule localization rewrite of clause `r2` is the following two clauses:

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

Here, the predicate `aux` is a new predicate: it does not appear in the original alphabet of predicates. As specified in the rule `r2-1`, this predicate is 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` tuple 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.

Problems with Set-Semantics in Distributed Setting.

While maintaining a view, derived tuples might need to be deleted from the view due to the deletion of base tuples. For instance, in the reachability example above, if a `link` tuple is deleted, one might need to delete some `reachable` tuples that are derived from it. In traditional, centralized incremental maintenance algorithms, such as DRed [8], one maintains a view by using set-semantics. That is, one does not keep track of the number of supporting derivations for any tuple. Then, whenever a tuple, p, is deleted, one eagerly deletes all the tuples that are supported by a derivation that contains p. Since some of the deleted tuples may be supported by alternative derivations that do not use p, DRed re-derives them in order to maintain a correct view.

It turns out that re-deriving tuples in a distributed setting is expensive due to high communication overhead, as demonstrated in [10]. A better approach is to use an evaluation algorithm that uses multiset-semantics to keep track of the number of supporting derivations of any tuple. So, whenever a tuple is deleted, such algorithm just needs to reduce its multiplicity by one, and whenever its multiplicity is zero, the tuple is deleted from the view.

However, guaranteeing termination when an algorithm uses multiset semantics is much harder, since tuples might be supported by infinitely many derivations. For example, in the reachability program above, if two nodes `a` and `b` are connected within a path that contains a cycle, one can derive the fact `reachable(a,b)` infinitely many times due to the recursion in the program. Therefore, if implemented in a naive way, an algorithm could easily diverge when keeping track of all supporting derivations.

3. BASIC PSN ALGORITHM FOR NON RECURSIVE PROGRAMS

We present our algorithm for distributed incremental view maintenance for non-recursive 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 PSN in the next section, before presenting an improved algorithm that provably ensures termination of recursive programs in Section 5.

3.1 System Assumptions

This paper makes two main assumptions about the model of our distributed system. The first assumption, following [11], 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. The second assumption is that messages are never lost during transmission, that is, a message eventually reaches its final destination. Here, we are not interested in the mechanisms of how the transmission is done, but we assume that any message is eventually received by the correct node specified by the location specifier `@`. Notice that, differently from previous work [10, 11], it is possible in our model that messages are reordered. That is, we do not assume that a message that is sent before another message has to necessarily arrive at its destination first[1]

The assumptions above are realistic for many systems, such as in networking or systems involving robots. For instance, without the bursty model, the links in a network could be changing constantly. Due to network propagation delays, no routing protocol would not be able to correctly 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, where new updates can appear at any moment. For the assumption that messages are never lost and eventually received, 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 enforcing that messages are not lost.

3.2 Definitions

An update is represented as a pair $\langle U, p\rangle$, where U is either the INS, denoting an insertion, or DEL, denoting a deletion, and p is a ground fact. We call an update of the form $\langle \text{INS}, p\rangle$ and $\langle \text{DEL}, p\rangle$ an *insertion update* and *deletion update*, respectively.

We write $\mathcal{U}$ to denote a multiset of updates. For instance, the following multiset of updates

$\mathcal{U} = \{\langle \text{INS}, p(@1, d)\rangle, \langle \text{DEL}, p(@2, a)\rangle, \langle \text{DEL}, p(@2, a)\rangle\}$,

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

As mentioned in Section 2, we will use multiset semantics. We use $\uplus$ as the multiset union operator, and $\setminus$ as the multiset minus operator. We write P to denote the multiset view for the predicate p, and ΔP to denote the multiset of updates to predicate p. We write P^ν to denote the updated view of p based on ΔP. P^ν can be computed from P and ΔP by union P with all the tuples inserted by ΔP and minus the tuples deleted by ΔP. For ease of presentation, we use the predicate name p^ν in places where we need to use the updated view. For instance, if the view of p is the multiset $\{p(a), p(a), p(b), p(c)\}$ and we use the multiset of update $\mathcal{U}$ shown above, the resulting view (P^ν) for p^ν is the multiset $\{p(b), p(c), p(d)\}$.

Delta-Rules.

The main task of the algorithm is to compute which tuples can be derived from the insertion updates and which tuples need to be deleted due to the deletion updates, given a multiset of insertions and deletions, $\mathcal{U}$, to base tuples. The main idea is that we can modify the rules in the corresponding program to do so. Consider,

[1]Message reordering manifests itself in several practical scenarios. For instance, in addition to reordering of messages buffered at the network layer, network measurements studies such as [15] have shown that packets may traverse different Internet paths for any two routers due to ISP policies, and in a highly disconnected environment such as in Robotics [4], messages from a given source to destination may traverse different paths due to available network connectivity during the point of transmission of each message.

for example, the rule p :- b_1, b_2 whose body contains two elements. There are the following three possible cases one needs to consider in order to compute the changes to the view of the predicate p when using this rule: Δp :- $\Delta b_1, b_2$, Δp :- $b_1, \Delta b_2$, and Δp :- $\Delta b_1, \Delta b_2$. The first two just takes 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, 17], we can simplify the delta-rules above by using the view 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 $1 \leq i \leq n$:

$\langle \text{INS}, h(\vec{t}) \rangle$:- $b_1^\nu(\vec{t_1}), \ldots, b_{i-1}^\nu(\vec{t_i}_{-1}), \Delta b_i(\vec{t_i}), b_{i+1}(\vec{t_i}_{+1}), \ldots, b_n(\vec{t_n})$

$\langle \text{DEL}, h(\vec{t}) \rangle$:- $b_1^\nu(\vec{t_1}), \ldots, b_{i-1}^\nu(\vec{t_i}_{-1}), \Delta b_i(\vec{t_i}), b_{i+1}(\vec{t_i}_{+1}), \ldots, b_n(\vec{t_n})$

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

As discussed in [8], we can prove following lemma stating that derivation are indeed computed only once using these delta rules.

LEMMA 1 (UNIQUE DERIVATION). *Given a multiset of updates, each of which is supported by a unique derivation, firing the delta-rules formalized as above generates each update supported by a unique derivation only once.*

PROOF. This follows from the way delta-rules are constructed. Whenever there are more than one update inserting (respectively, deleting) tuples appearing in the body of the same delta rule, only the update whose tuple appears in the right-most (respectively, left-most) position can fire that delta-rule. □

3.3 PSN Algorithm

We propose Algorithm 1 for maintaining incrementally a distributed view given a *DDlog* program using an enhanced version of the original pipelined evaluation strategy [11]. Since all tuples are stored according to the `@` operator, we can use a single multiset $\mathcal{K}$ containing the union of views of all the nodes in the system. From $\mathcal{K}$ one can figure out the specific node where the data is stored by examining the attribute annotated by the `@` operator. 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 view of all nodes in the system, one marked with ν and another without ν. The execution of one node of the system is specified by one iteration of the while-loop in Algorithm 1. In line 2, one *picks non-deterministically* an update from $\mathcal{U}$ which is processed next. However, one is only allowed to pick a deletion update if the tuple being deleted is present in the view $\mathcal{K}$. This is specified by the operation $removeElement(\mathcal{K})$. This operation avoids tuples 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, one uses the update picked to *fire* delta-rules and to 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 remaining lines, one *commits* the changes to the view without ν 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 can formally prove that Algorithm 1 always terminates on non-recursive programs.

LEMMA 2. *For non-recursive programs, PSN executions always terminate.*

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 ⟨INS, p(t⃗)⟩
4:       P^ν = P ⊎ {p(t⃗)}
5:    if δ is a deletion update ⟨DEL, p(t⃗)⟩
6:       P^ν = P \ {p(t⃗)}
7:    if δ is an insertion update ⟨INS, b(t⃗)⟩
8:       execute all insertions delta-rules for b:
9:          ⟨INS, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
10:   if δ is a deletion update ⟨DEL, b(t⃗)⟩
11:      execute all deletion delta-rules for b:
12:         ⟨DEL, 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 ⟨INS, p(t⃗)⟩
17:      P = P ⊎ {p(t⃗)}
18:   if δ is a deletion update ⟨DEL, p(t⃗)⟩
19:      P = P \ {p(t⃗)}
20: end while
```

PROOF. To show termination we rely on the fact that the dependency graph for a non-recursive program contains no cycles, that is, it is a directed acyclic graph. First, we order the predicate names in the dependency graph in a sequence by using any of the graph's topological sorts $\mathcal{S}$. Then given a set of $\mathcal{U}$ of updates at the beginning of an iteration of the while-loop that remain to be processed by Algorithm 1, we construct a state-tuple associated to $\mathcal{U}$ as follows: for the i^{th} position of the state-tuple, we count the number of updates inserting or deleting tuples whose predicate name is the same as the predicate name appearing at the i^{th} position of $\mathcal{S}$. We can show that after an iteration of Algorithm 1's while-loop the state-tuple reduces its value with respect to the lexicographical ordering, which is well-founded since their are finitely many predicate names in the program. After an iteration, an update of a tuple whose predicate name appears at the i^{th} position in $\mathcal{S}$ is replaced by a set of updates of predicates that appear at the $i^{th} + m$ position, where $m > 0$. Therefore, the value of the resulting state-tuple decreases. Moreover, since we are assuming a language with finitely many symbols and we assume that Algorithm 1 starts with a finite number of updates, it is not possible to create indefinitely many different updates. Hence, Algorithm 1 terminates. □

An Example Execution.

We illustrate an execution of Algorithm 1 with a simple example adapted from [8], which specifies two and three hop reachability:

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

Here only the predicate `link` is a base tuple. Furthermore, assume that the view is as given below, where we elide the predicate names and the `@` symbols. For example, the tuples `link(@a,b)` and `hop(@a,c)` are in the view.

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

Notice that in the view above some tuples appear with multiplicity greater than one, that is, there are more than one derivation supporting such tuples. Assume that there is the following changes to the set of base tuples `link`:

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

Algorithm 1 first *picks* an update non-deterministically, for instance, the update $u = \langle \text{INS}, \texttt{link(d,f)} \rangle$, which causes an inser-

tion of the tuple `link(d,f)` to the table marked with ν. Then, it uses u to propagate new updates by *firing* rules, which in this case creates a single insertion update: $\langle \text{INS}, \texttt{hop(d,g)} \rangle$. Finally, one *commits* the change due to the update u in the table without ν. Hence the new set of updates and the new view of the link table are as follows:

$\mathcal{U}$ = {$\langle$INS,hop(d,g)$\rangle$, $\langle$INS,link(a,f)$\rangle$, $\langle$DEL,link(a,b)$\rangle$}
Link = {(a,b),(a,d),(d,c),(b,c),(c,h),(f,g),(d,f)}

Asynchronous Execution.

As we mentioned earlier, 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 view, 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. Because of the *non-deterministic* pick of an update in line 2 and rule localization, we can show that any concurrent execution of Algorithm 1 yields the same view as some synchronized run over all nodes. More specifically, rule localization ensures that all the predicates needed for firing a delta-rule reside in the local database. Furthermore, each iteration of Algorithm 1 only considers one update at a time. New updates received from other nodes will be queued up for consideration after the completion of the current iteration. In other words, when two nodes execute one iteration of Algorithm 1 concurrently, each node only access its local state. Therefore, the interleaving of one iteration of Algorithm 1 by two nodes produces the same result as a sequentialized run of the two iterations.

In contrast to the algorithms in the literature, one does not process all the updates involving `link` tuples before processing `hop` or `tri_hop` tuples. In fact, in the next iteration of Algorithm 1, one is allowed to pick the update $\langle \text{INS}, \texttt{hop(d,g)} \rangle$ although there are insertions and deletions of `link` tuples still to be processed. This asynchronous behavior makes the correctness proof for Algorithm 1 much harder. In the synchronized algorithms proposed in the literature one could rely on the following invariant: in an iteration one only processes updates that insert or delete tuples that are supported by derivations of some particular height. This is no longer the case for Algorithm 1 and therefore, we need to proceed our correctness proofs differently in the next section.

4. CORRECTNESS OF BASIC PSN

To prove its correctness, we first formally define the operational semantics of Algorithm 1 in terms of state transitions. 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 PSN can be transformed into an execution using SN.

4.1 Operational Semantics for Algorithm 1

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

pick – One picks non-deterministically one update, u, that is not a deletion of a tuple that is not (yet) in the view, from the multiset of updates $\mathcal{U}$. If u is an insertion of predicate p, p^ν is inserted into the updated view 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, one commits the change to the view caused by u: if u is an insertion update of a tuple p, p is inserted into the view P; otherwise, if it is a deletion update of p, p is deleted from the view P. This basic command is used in lines 16–19 in Algorithm 1.

We formalize the operational semantics of Algorithm 1 in terms of state transitions. A state 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 views of derivable predicates, and it contains both the view (P) and the updated view (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 are not used to update the view yet. At the end of the execution, $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{E}$ should be empty signaling that all updates have been processed, and all the views P in $\mathcal{K}$ are the final view of the system.

The transition rules specifying the operational semantics for the basic commands are shown in Figure 1. 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 view in $\mathcal{K}$: $p^\nu(\vec{t})$ is inserted into $\mathcal{K}$ if U is INS, otherwise it is deleted from $\mathcal{K}$ if U is DEL. 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 view, $\mathcal{K}$, 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 view. Similar to the rules for $pick$, they either insert into or delete from the updated view P a fact $p(\vec{t})$.

A *computation run* using a program $\mathcal{R}$ is a valid sequence of applications of these transition rules defined in Figure 1. We call the first state of a computation run the initial state and its last state the resulting state.

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 state $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{E}$ are empty.*

$$\frac{\langle \text{INS}, p(\vec{t})\rangle \in \mathcal{U}}{\langle \mathcal{K},\mathcal{U},\mathcal{P},\mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \uplus \{p^{\nu}(\vec{t})\}, \mathcal{U} \setminus \{\langle \text{INS}, p(\vec{t})\rangle\}, \mathcal{P} \uplus \{\langle \text{INS}, p(\vec{t})\rangle\}, \mathcal{E}\rangle}\ [pick_I] \qquad \frac{\langle \text{INS}, p(\vec{t})\rangle \in \mathcal{E}}{\langle \mathcal{K},\mathcal{U},\mathcal{P},\mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \uplus \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{INS}, p(\vec{t})\rangle\}\rangle}\ [commit_I]$$

$$\frac{\langle \text{DEL}, p(\vec{t})\rangle \in \mathcal{U} \quad \text{and} \quad p^{\nu}(\vec{t}) \in \mathcal{K}}{\langle \mathcal{K},\mathcal{U},\mathcal{P},\mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{p^{\nu}(\vec{t})\}, \mathcal{U} \setminus \{\langle \text{DEL}, p(\vec{t})\rangle\}, \mathcal{P} \uplus \{\langle \text{DEL}, p(\vec{t})\rangle\}, \mathcal{E}\rangle}\ [pick_D] \qquad \frac{\langle \text{DEL}, p(\vec{t})\rangle \in \mathcal{E}}{\langle \mathcal{K},\mathcal{U},\mathcal{P},\mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{p(\vec{t})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{DEL}, p(\vec{t})\rangle\}\rangle}\ [commit_D]$$

$$\frac{u \in \mathcal{P} \quad \text{and} \quad \mathcal{F} = \mathit{firRules}(u, \mathcal{K}, \mathcal{R})}{\langle \mathcal{K},\mathcal{U},\mathcal{P},\mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K}, \mathcal{U} \uplus \mathcal{F}, \mathcal{P} \setminus \{u\}, \mathcal{E} \uplus \{u\}\rangle}\ [\mathit{fire}]$$

Figure 1: Operational Semantics for Basic Commands

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

```
 1: while U.size > 0 do
 2:   for all insertion updates u = ⟨INS, h(t⃗)⟩ in U do
 3:     I_h.insert(h(t⃗))
 4:   end for
 5:   for all deletion updates u = ⟨DEL, 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 ⟨INS, b(t⃗)⟩
14:       execute all insertions delta-rules for b:
15:         ⟨INS, h⟩ :- b_1^ν, ..., b_{i-1}^ν, Δb, b_{i+1}, ..., b_n
16:     if δ is a deletion update ⟨DEL, b(t⃗)⟩
17:       execute all deletion delta-rules for b:
18:         ⟨DEL, 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
```

A PSN-iteration corresponds to an iteration of the outermost loop of Algorithm 1, as one picks only one update from $\mathcal{U}$ (lines 2–6), fires delta-rules using this update (lines 7–15), and then commits the change to the view (lines 16–19).

4.2 Correctness of an SN Evaluation

We define an incremental view maintenance algorithm based on synchronous 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.

4.2.1 A Synchronous SN Algorithm

We define a synchronous SN algorithm as shown in Algorithm 2 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 each iteration, each nodes process all available updates, and propagate generated new updates to each other. Then, 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.

Interestingly, the operational semantics for Algorithm 2 can also be defined in terms of the three basic commands: *pick*, *fire*, and *commit*. One picks (lines 2–10) all the updates from the multiset of updates $\mathcal{U}$, uses them to fire delta-rules (lines 11–22) creating new updates, which are inserted in $\mathcal{U}$ (line 23), and then commits the changes to the view (lines 24–26). Algorithm 2 enforces that all updates that are created at i^{th} iteration are necessarily processed in the $i^{th}+1$ iteration.

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 state $\mathcal{U}$.*

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

An SN-iteration corresponds exactly to an iteration of the outermost loop in Algorithm 2.

4.2.2 Proof of Correctness

Definitions.

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 tuples in $\mathcal{U}$. Given a program $\mathcal{P}$, let V be the view of a program $\mathcal{P}$ given the set of base facts E, and let V^{ν} be the view 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 tuples 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 tuples 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 multiset resulting from union of all multisets $\mathcal{U}[k]$ such that $i \leq k \leq j$.

Since Algorithm 2 uses multiset semantics, we need to be careful with the multiplicity of tuples as these need also to be maintained correct. In our proofs, we keep track of the multiplicity of tuples by distinguishing between different occurrences of the same tuple in the following form: we label different occurrences of the same base tuple with different natural numbers and label each occurrence of the same derived tuple with the derivation supporting it. For instance, consider the program from Section 2.1:

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

The view for this program is the multiset of annotated tuples $V = \{\mathtt{s}^{\Xi_1}, \mathtt{s}^{\Xi_2}, \mathtt{t}^{\Xi_3}, \mathtt{q}^1, \mathtt{u}^1\}$. The 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 $\mathtt{q}^1$ and the latter is a derivation

tree also with a single leaf but with the base tuple $\mathtt{u}^1$. If we, for example, delete the base tuple $\mathtt{u}^1$, then the resulting view changes to $V^\nu = \{\mathtt{s}^{\Xi_1}, \mathtt{q}^1\}$, where the difference set is

$\Delta = \{\langle \text{DEL}, \mathtt{u}^1\rangle, \langle \text{DEL}, \mathtt{s}^{\Xi_2}\rangle, \langle \text{DEL}, \mathtt{t}^{\Xi_3}\rangle\}$,

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

We elide these annotations whenever they are clear from the context.

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. [11] in defining the correctness of declarative networking protocols.

DEFINITION 8 (CORRECTNESS). *We say that an algorithm correctly maintains the view if it takes as input, a program $\mathcal{P}$, the view 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 view when the algorithm finishes is the same as V^ν, which is the view of $\mathcal{P}$ given the set of facts $E \uplus I^t \setminus D^t$.*

Algorithm 2 computes a multiset of updates $\mathcal{U}$ that are applied to the view V. Ideally, we want to show that the multiset of updates computed by Algorithm 2 is the same as Δ, which is the difference between the initial V and the desired final result V^ν. The correctness proof of Algorithm 2 is composed of two parts: (1) all the updates generated by Algorithm 2 are in Δ (Algorithm 2 is sound); and (2) Algorithm 2 generates all the updates in Δ (Algorithm 2 is complete).

Soundness of Synchronous SN.

We first show that Algorithm 2 does not perform more updates to the view than what's specified in Δ. Given a terminating execution of Algorithm 2, let's assume that the execution consists of n iterations. Intuitively, the soundness statement would require that $\mathcal{U}[0, n] \subseteq \Delta$. However, this is not true. Consider the following program with two clauses: p :- q, r and q :- s. Assume that the original view V is $\{s, q\}$ and that one provides the updates $\{\langle \text{INS}, r\rangle, \langle \text{DEL}, s\rangle\}$. Then the view $V^\nu = \{r\}$, $\Delta = \{\langle \text{INS}, r\rangle, \langle \text{DEL}, s\rangle, \langle \text{DEL}, q\rangle\}$. After the first iteration of Algorithm 2, the resulting set of new updates $\mathcal{U}[1] = \{\langle \text{INS}, p\rangle, \langle \text{DEL}, q\rangle\}$. The update $\langle \text{INS}, p\rangle$ is not in Δ but in $\mathcal{U}[1]$. Notice that $\langle \text{INS}, p\rangle$ is supported by a proof that uses the base fact r, which is inserted; and the fact q, which is supported by a proof that uses a deleted fact s. The deletion of s needs some more iterations to "catch up" and correct the unsound insertion of p.

We classify an update u 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). In the example above, $\langle \text{INS}, p\rangle$ is a conflicting updated because it is supported by r, which is inserted and s, which is deleted. One key observation is that Algorithm 2 may compute more updates than those in Δ. These extra updates are all conflicting updates. We need to show that the effects of all conflicting updates eventually cancel each other out.

The following lemma formalizes the intuition that updates that are needed to change V to V^ν are all non-conflicting updates.

LEMMA 9. *All updates in Δ are non-conflicting.*

PROOF. Consider by contradiction that an insertion update $u \in \Delta$ of the tuple p is conflicting. Then p is supported by a tuple q that is deleted from the view V. This is a contradiction because then p is no longer derivable in V^ν; and therefore, the insertion, u, of p could not have been in Δ.

Similarly, assume that a deletion update $u \in \Delta$ of the tuple p is conflicting. Then p is supported by a tuple q that is inserted to V. Again, we have a contradiction, since then p could not have been in V; and hence u could not have been in Δ. □

The following lemma states that the non-conflicting updates (updates that are supported only by insertion updates or only by deletion updates) generated at each iteration by the algorithm, are necessary to change V to V^ν.

LEMMA 10 (SOUNDNESS OF NON-CONFLICTING UPDATES). *Let $\widehat{\mathcal{U}}$ be the multiset of non-conflicting updates in a multiset of updates $\mathcal{U}$. Then for any iteration i, the multiset $\widehat{\mathcal{U}}[i] \subseteq \Delta$.*

PROOF. We proceed by induction on the number of iterations i. For the base case, we have $\widehat{\mathcal{U}}[0] = I \uplus D = \Delta[0] \subseteq \Delta$.
For the inductive case, consider $i = j + 1$ and the inductive hypothesis $\widehat{\mathcal{U}}[k] \subseteq \Delta$ for all $k \leq j$. Assume that $u = \langle \text{INS}, p^\Xi\rangle \in \widehat{\mathcal{U}}[j+1]$, and it is computed by using a delta-rule of the rule p :- $b_1, \ldots, b_n$ and the tuples or the insertion of tuples of the form $b_1^{\Xi_1}, \ldots, b_n^{\Xi_n}$. Since u is non-conflicting, all smaller derivations Ξ_is are also non-conflicting. Hence from the inductive hypothesis, all the insertions used by Ξ_is, including any insertion of $b_j^{\Xi_j}$, belong to Δ. Hence the tuples $b_1^{\Xi_1}, \ldots, b_n^{\Xi_n}$ belong to V^ν, and therefore by using the same rule above, there is an insertion of the tuple p^Ξ in V^ν, that is $\langle \text{INS}, p^\Xi\rangle \in \Delta$. The case for deletion follows similarly. □

Now, we turn our attention to the conflicting updates. We write $\bar{u}$ to denote the complement update of u. If u is an insertion (respectively, deletion) update of a tuple p, then $\bar{u}$ is a deletion (respectively, insertion) update of the same tuple p. We show that conflicting updates always exist in complementary pairs; and moreover, the insertion update is always generated at an iteration that is no later than the complementary deletion update.

LEMMA 11 (PAIRING OF CONFLICTING UPDATES). *For any conflicting update $u \in \mathcal{U}[i]$, there is exactly one update $\bar{u} \in \mathcal{U}[j]$, for some j, that is supported by the same derivation. If u is an insertion update then $i \leq j$, and if u is a deletion update then $i \geq j$.*

PROOF. Let us first prove that conflicting insertion updates are computed first. Given a conflicting deletion update $\langle \text{DEL}, p\rangle$ that is generated at iteration i, it must be the case that a delta-rule $\langle \text{DEL}, p\rangle$:- $b_1^\nu, \ldots, b_{m-1}^\nu, \Delta b_m, b_{m+1}, \ldots b_n$ is fired. By the definition of conflicting updates, one of the tuples b_i in the body is supported by a tuple that must be inserted. Since the body of the rule above can only be satisfied when b_i is inserted, the insertion of b_i must have been necessarily picked before or at the iteration i, firing another delta-rule similar to the rule above. Hence, the insertion update for the tuple p is created before or at iteration i.

Next we show that for any conflicting insertion update, a complementary deletion update is generated at the same or in a later iteration. Given an insertion update $u \in \mathcal{U}[i]$. Let m be the minimal height among all the subtrees of the derivation supporting the tuple in u that contain a tuple, b_i, that is deleted. In exactly m iterations, the corresponding deletion delta-rule is going to be fired using the deletion update for b_i, generating a deletion update $\bar{u}$ with a tuple with same supporting proof. □

Completeness of Synchronous SN.

Now we prove that all the updates in Δ are generated by Algorithm 2. The following lemma states that all updates in Δ that are supported by a derivation of height i has already been computed by Algorithm 2 at an iteration that is no later than i.

LEMMA 12 (COMPLETENESS). *For any i, $\Delta[i] \subseteq \mathcal{U}[0, i]$.*

PROOF. By induction on the height of proofs.
Base case $i = 0$: $\Delta[0] = I \uplus D = \mathcal{U}[0] = \mathcal{U}[0, 0]$.
Inductive case $i = j+1$: By induction hypothesis, we know that all $\Delta[k]$, where $k < j + 1$, have been computed. Now, we show that all updates in $\Delta[j + 1]$ are contained in $\mathcal{U}[0, j + 1]$. Assume that $\langle \text{INS}, p^{\Xi} \rangle \in \Delta[j + 1]$ and assume that p^{Ξ} is supported in the view V^{ν} by using rule p :- $b_1, \ldots, b_n$ called r and tuples $b_1^{\Xi_1}, \ldots, b_n^{\Xi_n}$ also in V^{ν}. We now show that a delta-rule of r is fired before the $j^{th} + 1$ iteration. Since $p^{\Xi} \notin V$, it means that some $b_i^{\Xi_i}$s do not belong to V, but belong V^{ν} (hence the insertion update). Since the insertion of Ξ is a derivation of height $j + 1$, the Ξ_is are derivations of height at most j. Hence, from the inductive hypothesis, it is the case that the insertions of the $b_i^{\Xi_i}$s have been previously derived and in the worst case the delta-rule for r is fired at the iteration j. However, in order to fire a delta-rule of r, we also need to make sure that Algorithm 2 does not delete any of the $b_i^{\Xi_i}$s. Since $\langle \text{INS}, p^{\Xi} \rangle$ is in Δ, it follows from Lemma 9 that Ξ is non-conflicting. So, no tuple $b_i^{\Xi_i}$s is supported by a tuple that is deleted and hence indeed none of the $b_i^{\Xi_i}$s are deleted by Algorithm 2. Therefore, $\langle \text{INS}, p^{\Xi} \rangle \in \mathcal{U}[0, j + 1]$. The case for deletion updates is similar. □

Notice that in the proof, we use invariants that relate the derivation height of the tuples to the iteration number of the while loop. This would not have been possible for Algorithm 1.

Correctness of Synchronous SN.

Combining the soundness and completeness result, we can finally show the correctness of Algorithm 2.

THEOREM 13 (CORRECTNESS OF SN). *Given a non-recursive* DDlog *program $\mathcal{P}$, a multiset of base tuples, E, a multiset of updates insertion updates I and deletion updates D to base tuples, such that $D^t \subseteq E \uplus I^t$, Algorithm 2 correctly maintains the view of the database when it terminates.*

PROOF. Because $\mathcal{P}$ is non-recursive, we know that both V and V^{ν} is finite; and therefore, Δ is also finite.

By the definition of the transition rules, given a complete run of Algorithm 2, the final view V_1 computed by Algorithm 2 is $V \uplus \mathcal{U}_I^t[0, n] \setminus \mathcal{U}_D^t[0, n]$, where n is the number of iterations of the execution, $\mathcal{U}_I$ denotes the insertions updates in $\mathcal{U}$, and $\mathcal{U}_D$ denotes the deletion updates in $\mathcal{U}$.

Let $\widehat{\mathcal{U}}$ denotes the non-conflicting updates in $\mathcal{U}$. By Lemma 10, $\widehat{\mathcal{U}}[0, n] \subseteq \Delta$. By Lemma 12, $\Delta \subseteq \mathcal{U}[0, n]$. By Lemma 9, $\Delta \subseteq \widehat{\mathcal{U}}[0, n]$. Therefore, $\Delta = \widehat{\mathcal{U}}[0, n]$. By Lemma 11, $V \uplus \mathcal{U}_I^t[0, n] \setminus \mathcal{U}_D^t[0, n] = V \uplus \widehat{\mathcal{U}}_I^t[0, n] \setminus \widehat{\mathcal{U}}_D^t[0, n]$. Since $V^{\nu} = V \uplus \Delta_I^t \setminus \Delta_D^t$, we can conclude that $V_1 = V^{\nu}$. □

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 view of the execution.

Definitions.

We write $s \xrightarrow{sn} (\mathcal{U})s'$ and $s \xrightarrow{psn} (\mathcal{U})s'$ to denote, respectively, an execution from state s to s' using an SN iteration and an PSN iteration. We annotate the updates used in the iterations in the parenthesis after the arrow. We write $s \xrightarrow{a} s'$ to denote an execution from s to s' using multiple SN iterations, when a is *sn*; or PSN iterations, when a is *psn*. We write $s \Longrightarrow s'$ to 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 $\sigma_2 \rightsquigarrow \sigma_1$.

Permuting PSN-iterations.

The following lemma states that permuting two PSN-iterations that are both insertion (deletion) updates leaves the final state unchanged.

LEMMA 14 (PERMUTATION – SAME KIND).
Given an initial state s,
$s \xrightarrow{psn} (\{\langle U, r_1 \rangle\})s_1 \xrightarrow{psn} (\{\langle U, r_2 \rangle\})s'$
$s \xrightarrow{psn} (\{\langle U, r_2 \rangle\})s_2 \xrightarrow{psn} (\{\langle U, r_1 \rangle\})s'$ *,where* $U \in \{\text{INS}, \text{DEL}\}$.

PROOF. We show the case where $U = \text{INS}$ for the $\rightsquigarrow$ direction, the other cases are similar. We need to show that the updates generated are the same no matter which insertion update is fired first.

Let's assume that the initial state $s = \langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset \rangle$.

Let $F_1 = \mathit{firRules}(\langle \text{INS}, r_1 \rangle, \mathcal{K} \uplus \{r_1^{\nu}\}, \mathcal{R})$,
$F_2 = \mathit{firRules}(\langle \text{INS}, r_2 \rangle, \mathcal{K} \uplus \{r_1, r_1^{\nu}, r_2^{\nu}\}, \mathcal{R})$.
Let $F_2' = \mathit{firRules}(\langle \text{INS}, r_2 \rangle, \mathcal{K} \uplus \{r_2^{\nu}\}, \mathcal{R})$,
$F_1' = \mathit{firRules}(\langle \text{INS}, r_1 \rangle, \mathcal{K} \uplus \{r_2, r_2^{\nu}, r_1^{\nu}\}, \mathcal{R})$.

In the first execution sequence, F_1 contains updates generated by firing delta-rules that contain Δr_1 in the body using the initial views with r_1^{ν} inserted, and F_2 contains updates generated by firing delta-rules that contain Δr_2 in the body using the views where r_1 is already inserted into the view.

In the second execution sequence, F_2' contains updates generated by firing delta-rules that contain Δr_2 in the body using the initial views with r_2^{ν} inserted, and F_1' contains updates generated by firing delta-rules that contain Δr_1 in the body from the state where r_2 is already inserted into the view.

We need to show that $F_1 \uplus F_2 = F_1' \uplus F_2'$.

Based on the definition of $\mathit{firRule}$, it is not hard to see that F_1' is a superset of F_1 because in the second execution sequence, r_2 is already inserted into the view before firing update to r_1. Similarly, F_2 is a superset of F_2'. Let us assume that $F_1' = F_1 \uplus F_1''$, and $F_2 = F_2' \uplus F_2''$. We just need to show that $F_1'' = F_2''$.

All updates in F_1'' are fired by rules that have Δr_1 and either r_2 or r_2^{ν} in the body. Without loss of generality, any update $u = \langle \text{INS}, q \rangle \in F_1''$ is created by firing delta-rules of the following two forms: u :- $\cdots, r_2^{\nu}, \cdots, \Delta r_1, \cdots$ or u :- $\cdots, \Delta r_1, \cdots r_2 \cdots$.

If it is the first case, then a corresponding delta-rule
u :- $\cdots, \Delta r_2, \cdots, r_1, \cdots$ will be fired when $\langle \text{INS}, r_2 \rangle$ is picked; and therefore, $\langle \text{INS}, q \rangle \in F_2''$.

For the second case, a corresponding delta-rule
u :- $\cdots, r_1^{\nu}, \cdots \Delta r_2 \cdots$ will be fired; and therefore $\langle \text{INS}, q \rangle \in F_2''$ also. Consequently, $F_1'' \subseteq F_2''$. We can use similar reasoning to show that $F_2'' \subseteq F_1''$. Combining the above two, $F_2'' = F_1''$. Therefore $F_1 \uplus F_2 = F_1' \uplus F_2'$. Finally, we can conclude that permuting two insertion updates leaves the final state unchanged. □

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 view. Furthermore, assume the updates $\{\langle \text{INS}, r_1 \rangle, \langle \text{DEL}, r_2 \rangle\}$. 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, once 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. It is proved in a similar way as Lemma 14. The detailed proof can be found in the Appendix A. The side condition that $r_1 \neq r_2$ captures the semantics of the pick command in that deletion updates are only picked in the tuples to be deleted are already in the view.

LEMMA 15 (PERMUTATION – DIFFERENT KIND). *Given and initial state s*

$s \xrightarrow{psn} (\langle \text{INS}, r_1\rangle)s_1 \xrightarrow{psn} (\langle \text{DEL}, r_2\rangle)\langle \mathcal{K}', \mathcal{U}' \uplus \Delta, \emptyset, \emptyset\rangle$

$\leftrightsquigarrow$

$s \xrightarrow{psn} (\langle \text{DEL}, r_2\rangle)s_2 \xrightarrow{psn} (\langle \text{INS}, r_1\rangle)\langle \mathcal{K}', \mathcal{U}', \emptyset, \emptyset\rangle$,

where $r_1 \neq r_2$ and Δ is a (possibly empty) multiset containing pairs of complementary conflicting updates.

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 view $\{q\}$, and the multiset of updates $\{\langle \text{INS}, r\rangle, \langle \text{DEL}, q\rangle\}$. If both updates are picked in a complete-iteration, then an insertion update, $\langle \text{INS}, p\rangle$, is created by firing the delta-rule $\langle \text{INS}, p\rangle$:- $\Delta r, q$ and using the insertion update $\langle \text{INS}, r\rangle$. Similarly a deletion update $\langle \text{DEL}, p\rangle$ is created by firing the delta-rule $\langle \text{DEL}, p\rangle$:- $r^\nu, \Delta q$ and the deletion update $\langle \text{DEL}, q\rangle$. However, if we break the complete-iteration into two PSN-iteration, the first picking the deletion update and the second picking the insertion update, then no delta-rule is fired. We prove the following:

LEMMA 16 (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 state.*

$s \Longrightarrow (\{u\} \uplus \mathcal{H})\langle \mathcal{K}', \mathcal{U}' \uplus F_1, \emptyset, \emptyset\rangle$

$\leftrightsquigarrow$

$s \Longrightarrow (\mathcal{H})\langle \mathcal{K}_2, \mathcal{U}' \uplus \{u\} \uplus F_1', \emptyset, \emptyset\rangle \xrightarrow{psn} (u)\langle \mathcal{K}', \mathcal{U}' \uplus F_2, \emptyset, \emptyset\rangle$

Where F_1 and F_2 only differs in pairs of complementary conflicting updates.

The proofs can be found in Appendix A. Lemma 16 actually give us for free, the ability to break a complete SN iteration into several PSN iterations.

Dealing with Conflicting Update Pairs.

Next, we prove that conflicting updates do not interfere with the final view when using PSN executions. Intuitively, all updates generated by firing delta-rules for conflicting updates are also conflicting updates. Furthermore, a pair of complementary conflicting updates generates pairs of complement conflicting updates. Consequently, a PSN execution that contains a pair of complement conflicting updates in its initial state can be transformed into another PSN execution that does not contain these updates and that the final states of the two executions are the same. The following lemma precisely states that. Detailed proofs can be found in Appendix A.

LEMMA 17. *Let $\Delta = \{\langle \text{INS}, p\rangle, \langle \text{DEL}, p\rangle\}$ be a multiset containing a pair of complementary conflicting updates, then*

$\langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle \xRightarrow{psn} s \quad \leftrightsquigarrow \quad \langle \mathcal{K}, \mathcal{U} \uplus \Delta, \emptyset, \emptyset\rangle \xRightarrow{psn} s.$

Note that 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 18 (CORRECTNESS OF PSN W.R.T. SN). *Let $s = \langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset\rangle$ be an initial state. Then for non-recursive programs:*

$s \xRightarrow{psn} \langle \mathcal{K}, \emptyset, \emptyset, \emptyset\rangle \leftrightsquigarrow s \xRightarrow{sn} \langle \mathcal{K}, \emptyset, \emptyset, \emptyset\rangle$.

PROOF. Given a PSN execution, we construct an SN execution by induction as follows: we use the first operation (Lemmas 14 and 15) to permute to the left all the PSN-iterations that pick one element in the initial state's $\mathcal{U}$ set. The resulting execution has all PSN-iterations in the same order as the first SN-iteration of an SN execution. After each permutation, we either generate new conflicting updates, or suppressed the generation of conflicting updates that is in the original execution. We apply Lemma 17 to transform the rest of the execution into a valid PSN execution, but leave the final state unchanged.

Next, we merge these PSN-iterations into an SN-iteration by applying the second operation (Lemma 16). Again, we need to apply Lemma 17 to transform the rest of the execution to account for the difference in conflicting updates.

We repeat the above process with the PSN sub-execution. This process will eventually terminate because there is a finite number of updates (conflicting and non-conflicting), with each iteration of the process, the sub-execution has fewer updates to generate.

For the converse direction, given an SN execution, we apply Lemma 16 repeatedly to split SN-iterations and obtain a PSN execution. Again we might need to apply the transformation described in Lemma 17 in order to construct valid executions. □

The above theorem states that the same derived tuples that are created by SN are also created by PSN and vice-versa. Hence, from Theorem 13, PSN is correct.

COROLLARY 19 (CORRECTNESS OF BASIC PSN). *Given a non-recursive* DDlog *program $\mathcal{P}$, a multiset of base tuples, E, a multiset of updates insertion updates I and deletion updates D to base tuples, such that $D^t \subseteq E \uplus I^t$, then Algorithm 1 correctly maintains the view of the database.*

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 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 Algorithm 2 use multiset-semantics. As a consequence, termination is not guaranteed, when they are used to maintain views 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 fact a(@0), neither Algorithm 1 nor Algorithm 2 will terminate because the propagation of insertion updates of q(@2) and p(@1) will not terminate. Recursively defined tuples could have infinite number of derivations because of cycles in the dependency graph. In other words, in the multiset-semantics, such tuples have infinite count. Neither Algorithm 1 nor Algorithm 2 have the ability to detect cycles.

One way to detect such cycles is proposed in [13] in a centralized setting. 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 view, one checks whether a newly derived tuple 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 tuple 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.

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 views and updates. A fact is now of the form $(p, \mathcal{S}, \mathcal{H})$, where p is a predicate, $\mathcal{S}$ 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 tuples that belong to a cycle in the derivation and therefore cause p to have an infinite count. Tuples have infinite count when they have a non-empty infinite count set. In the example above, the view of the nodes would be:

```
{(a,∅), (p,{a},∅), (q,{p,a},∅), (p,{a,p,q},{p}),...}
```

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

In order to maintain correctly the view, we adapt the transition rules accordingly. A summary of the rules are shown in Figure 2. Each *pick* rule in Figure 1 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 $pick_I$ and $pick_D$, one needs to first check whether the tuple is supported by a derivation tree that has a cycle. That is, one needs to check 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 view 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 tuples with infinite count need to be avoided. Given an update $\langle U, (b_i, \mathcal{S}_i, \mathcal{H}_i)\rangle$ inserting or deleting a tuple, in addition to computing all updates that are propagated from this update, we also construct 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 $b_1^\nu, \ldots, b_i^\nu, \Delta b_i, b_{i+1}, \ldots, b_n$ and the facts $(b_i, \mathcal{S}_i, \mathcal{H}_i)$ where $1 \leq i \leq n$, 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 tuple with infinite count is not resend. To do so, we only add the update $\langle U, (p, \mathcal{S}_p, \mathcal{H}_p)\rangle$ to the multiset of updates $\mathcal{U}$ if $p \notin \mathcal{H}_p$. That is, it is not part of cycle that has been already computed.

Returning to the previous example, when the update inserting tuple 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 p(@1)$\in \mathcal{S}$, node 1 detects the cycle in the derivation and adds the tuple 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, we can guarantee the termination of PSN using the derivation set and infinite count set on any recursive program, which is stated as follows.

THEOREM 20 (FINITENESS OF PSN THAT DETECTS CYCLES). *Let $\mathcal{S}$ be an initial state and $\mathcal{R}$ be a* DDlog *program. Then all PSN executions using $\mathcal{R}$ and from $\mathcal{S}$ have finite length.*

PROOF. Since we are assuming finite signature with no function symbols, there is a finite number N of facts in a system. We use a tuple with N elements, called state tuple, described next and the lexicographical ordering among them to show termination. Given a state of the system, the i^{th} element of the state tuple contains the the number of updates $\langle U, (p, \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U}$, such that $i = |\mathcal{S}|$, where $|\mathcal{S}|$ is the number of elements in $\mathcal{S}$. This ordering is clearly well founded. It is easy to show by induction on the length of runs that there cannot be any update whose associated derivation set $\mathcal{S}$ has more than N elements, since whenever a cycle is detected, an update is not resend. Only when the set of updates is empty, $\mathcal{U} = \emptyset$, can the least state tuple be reached. For any update message $u = \langle U, (p, \mathcal{S}, \mathcal{H})\rangle$, we denote $|u|$ as the number of elements in the multiset $\mathcal{S}$.

We show that the value of the state tuple reduces with respect to the lexicographical ordering after any PSN-iteration. After a PSN-iteration, there are two possible ways that the multiset of updates $\mathcal{U}$ is changed. The first case is when the picked update, u, does not contain a cycle. Then whenever a rule is fired, an update, u', is propagated such that the $|u| < |u'|$ since at least the tuple in u is inserted into the derivation set of u'. Then the update u' is inserted in the set $\mathcal{U}$, while the update u is removed from it. Therefore, the value of the i^{th} element in the state tuple, where $i = |u|$, is reduced by one, while all the values of the elements appearing before are untouched. The second case is when the update propagated is in the set $\mathcal{H}$ of tuples with infinite count. In this case, the update is not propagated and the total number of elements in $\mathcal{U}$ is reduced by one. Therefore, the value of the state tuple associated to the resulting state is also reduced. □

COROLLARY 21. *The PSN algorithm that detects cycles always terminates.*

Correctness for PSN that Detects Cycles.

Finally, we need to prove that the PSN algorithm that detects cycles maintains views correctly in the presence of recursive programs. The proofs follow the same steps as the proof for the correctness of the basic PSN algorithm in Section 4. First, we extend the basic SN algorithm (Algorithm 2) to deal with annotations for derivation and infinite count sets by using the new transition rules in Figure 2. Then, we prove that the extended SN algorithm is correct. Next, we relate PSN executions to SN executions.

However we need to revisit the definition of correctness. We have shown in the beginning of this section that the multiset semantics for recursive programs include tuples with infinite counts. That means that the view V and V^ν could be infinite, which implies that the updates that have to be computed (Δ) could be infinite as well. The definition for correctness only makes sense when Δ is finite, since no terminating programs can compute infinite set of updates. What the cycle-detection mechanism really does is to represent the infinite number of derivations for a recursive tuple by

$$\frac{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U} \quad \text{and} \quad p(\vec{t}) \in \mathcal{S} \quad \text{and} \quad \mathcal{H}' = \mathcal{H} \cup \{p(\vec{t})\}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \uplus \{(p^{\nu}(\vec{t}), \mathcal{S}, \mathcal{H}')\}, \mathcal{U} \setminus \{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H}')\rangle\}, \mathcal{E}\rangle} \; [pick_I^1]$$

$$\frac{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U} \quad \text{and} \quad p(\vec{t}) \notin \mathcal{S}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \uplus \{(p^{\nu}(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U} \setminus \{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{E}\rangle} \; [pick_I^2]$$

$$\frac{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U} \quad \text{and} \quad p(\vec{t}) \in \mathcal{S} \quad \text{and} \quad \mathcal{H}' = \mathcal{H} \cup \{p(\vec{t})\}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{(p^{\nu}(\vec{t}), \mathcal{S}, \mathcal{H}')\}, \mathcal{U} \setminus \{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H}')\rangle\}, \mathcal{E}\rangle} \; [pick_D^1]$$

$$\frac{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{U} \quad \text{and} \quad p(\vec{t}) \notin \mathcal{S}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{(p^{\nu}(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U} \setminus \{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{P} \uplus \{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}, \mathcal{E}\rangle} \; [pick_D^2]$$

$$\frac{u \in \mathcal{P} \quad \text{and} \quad \mathcal{F} = \mathit{firRules}(u, \mathcal{K}, \mathcal{R})}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K}, \mathcal{U} \uplus \mathcal{F}, \mathcal{P} \setminus \{u\}, \mathcal{E} \uplus \{u\}\rangle} \; [\mathit{fire}]$$

$$\frac{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{E}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \uplus \{(p(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{INS}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}\rangle} \; [commit_I]$$

$$\frac{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle \in \mathcal{E}}{\langle \mathcal{K}, \mathcal{U}, \mathcal{P}, \mathcal{E}\rangle \longrightarrow_{\mathcal{R}} \langle \mathcal{K} \setminus \{(p(\vec{t}), \mathcal{S}, \mathcal{H})\}, \mathcal{U}, \mathcal{P}, \mathcal{E} \setminus \{\langle \text{DEL}, (p(\vec{t}), \mathcal{S}, \mathcal{H})\rangle\}\rangle} \; [commit_D]$$

Figure 2: Operational semantics for the basic commands that detect cycles

one derivation that contains only one cycle. We revise the definition for correctness accordingly to reflect the fact that the standard resulting view V^{ν} that we compare against is a finite multiset view where a tuple that would have had infinite number of derivations in traditional fixed-point semantics now has a finite number of representative derivations. For instance, in a centralized setting, the semi-naïve evaluation algorithm described in [13] computes such a finite (multiset) view for recursive programs.

Then in the proof of correctness of SN executions, we add a new case when tuples with infinite counts are derived, that is, when they are supported by a derivation with a single cycle. This is indeed the case for any SN execution as the new *fire* rule does not propagate new updates when such updates are processed.

Finally, the proofs that relate a PSN execution to an SN execution remain almost the same except that we have to consider attaching annotations to tuples and updates; and that the termination argument for PSN is different. The transformations used in that proof continue to be valid when using the transition systems in Figure 2.

COROLLARY 22 (CORRECTNESS OF PSN). *Given any* DDlog *program $\mathcal{P}$, a multiset of base tuples, E, a multiset of updates insertion updates I and deletion updates D to base tuples, such that $D^t \subseteq E \uplus I^t$, then the PSN algorithm that detects cycles correctly maintains the view of the database.*

6. RELATED WORK

Earlier works on incremental view maintenance focus on Datalog programs in a centralized setting ([8, 6] to list a few). Whereas, our work focuses on designing efficient algorithms for maintaining views incrementally in a distributed setting. Compared to the traditional DRed algorithm proposed in [8], our algorithm uses multiset-semantics, while DRed uses set-semantics. As previously discussed, the use of set-semantics requires DRed to re-deriving tuples, a step which involves high communication overhead and makes the use of DRed impractical in a distributed setting. Since DRed is meant to work in a centralized setting, so it is a synchronous algorithm, where updates created in one iteration are necessarily processed in the following iteration. It is not clear to us whether DRed can be adapted to a distributed setting that we consider here to produce correct results and at the same time efficient.

In recent years, there has been several works tackling the problem of evaluating Datalog-like programs, and maintaining views in a distributed setting. Our work is based on the original proposal of PSN evaluation [11]. We extended the original proposal in several ways. First, Loo *et al.* considered only linear recursive terminating Datalog programs. We consider the complete Datalog language including non-linear recursive programs. Second, we relaxed the assumptions in the original proposal: instead of assuming that the transmission channels are FIFO, we do not make any assumption about the order in which updates are processed. The most important improvement is that the PSN algorithm proposed in this paper is proven to terminate and maintain views correctly. As pointed out in our previous work [14], the PSN algorithm as presented in [11] may produce unsound results and the use of the count algorithm [8] leads to non-termination.

Similar to our approach, Liu *et al.* in [10] propose a PSN algorithm that attaches annotations to tuples with provenance information [5] to track duplicate derivations and avoid non-termination due to cycles in recursive programs. However, Liu *et al.* only track the base tuples used in the derivation. While our *derivation set* contains all facts (including base and intermediate derived) used for each derivation. Using only base tuples, 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 the remaining updates. When messages are processed out of order, the algorithm proposed in [10] yields unsound results. We show such an example in Appendix B. Finally, the algorithm is only experimentally evaluated but not formally proven correct.

In contrast to our approach, where we annotate tuples with the set of facts used to derive them, MELD [4] simply attaches to tuples the height of the derivation supporting them. They are able to make many optimizations to the way in which updates are processed. For instance, they do allow nodes to pick a deletion update of a tuple that is supported by a derivation of a height greater than the derivation of the same tuple appearing in the view. However, simply attaching the height of derivations to tuples is not enough to detect cycles in derivations and therefore it is not enough to avoid divergence by itself. They address this problem by enforcing the synchronization among nodes, that is, not allowing nodes to compute until they receive the response from other nodes during deletion. 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 tuples. We believe that their work can

directly leverage the results in this paper.

In an attempt to generalize Loo *et al.*'s work [11], Dedalus [3] 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 any semantics in the Datalog literature. Depending on whether a message is lost or not, the final views 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.

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 consequences resulting from inserting a formula, we are interested in efficiently maintaining a set of consequences that was previously derived. It is not clear how their approach can be used to update consequence when a sequence of insertions and deletions are made to the knowledge base.

7. CONCLUSIONS AND FUTURE WORK

This paper presents techniques for incrementally updating views for distributed recursive Datalog programs in the presence of insertions and deletions of base tuples. Our PSN algorithm improves upon existing techniques in the following ways. First, it is more bandwidth efficient than DRed [8], since it avoids unnecessary deletions and rederivations. Second, unlike its predecessors [11, 10] the algorithm presented in this paper maintains views correctly for general recursive programs, even in the presence of message reordering. By annotating tuples with information about its derivation, our algorithm can detect cycles in recursive programs. Most importantly, we prove that our PSN algorithm terminates and that it maintains views correctly.

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

In particular, our research group is interested in formal verification of implementations of networking protocols prior to actual deployment in declarative network setting [18, 19]. 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. Datalog programs with arbitrary functions symbols may not terminate. We are investigating if we can extend existing analysis techniques [9] 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 views incrementally in the present of aggregate functions. We are looking into adapting existing work, such as [16] in incremental view maintenance in a centralized setting to fit our needs.

8. 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] 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, Dec 2009.

[4] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IROS*, pages 2794–2800. IEEE, 2007.

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

[6] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *SIGMOD Rec.*, 24(2):328–339, 1995.

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

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

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

[10] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, pages 1108–1119, 2009.

[11] 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*, 2006.

[12] N. P. Lopes, J. A. Navarro, A. Rybalchenko, and A. Singh. Applying prolog to develop distributed systems. In *ICLP*, 2010.

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

[14] V. Nigam, L. Jia, A. Wang, B. T. Loo, and A. Scedrov. An operational semantics for network datalog. In *LAM'10*, 2010.

[15] V. Paxson. End-to-end routing behavior in the internet. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 25–38, New York, NY, USA, 1996. ACM.

[16] R. Ramakrishnan, K. A. Ross, D. Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SLP*, pages 204–218, 1994.

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

[18] A. Wang, P. Basu, B. T. Loo, and O. Sokolsky. Declarative network verification. In *11th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2009.

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

APPENDIX

A. PROOFS FOR LEMMAS RELATED TO CORRECTNESS

Proof of Lemma 15.

PROOF. We show the $\rightsquigarrow$ direction. The reasoning is symmetric for the reverse transformation. Let

$F_1 = firRules(\langle \text{INS}, r_1\rangle, \mathcal{K} \uplus \{r_1^\nu\}, \mathcal{R})$,
$F_2 = firRules(\langle \text{DEL}, r_2\rangle, \mathcal{K} \uplus \{r_1, r_1^\nu\} \setminus \{r_2^\nu\}, \mathcal{R})$,
$F_2' = firRules(\langle \text{DEL}, r_2\rangle, \mathcal{K} \setminus \{r_2^\nu\}, \mathcal{R})$,
$F_1' = firRules(\langle \text{INS}, r_1\rangle, \mathcal{K} \setminus \{r_2, r_2^\nu\} \uplus \{r_1^\nu\}, \mathcal{R})$.

In the first execution sequence, F_1 contains all insertion updates created from the initial view by firing insertion delta-rules that contain Δr_1 in their body. Similarly, F_2 contains all the deletion updates created by firing deletion delta-rules that contain Δr_2 in their body, with r_1 inserted into the initial view.

In the second execution sequence, on the other hand, F_2' contains all the deletion updates created from the initial view by firing deletion delta-rules that contain Δr_2 in their body. F_1' contains all the insertion delta-rules that contain Δr_1 in their body, with r_2 deleted from the view.

We would like to show that $F_1 \uplus F_2 = F_1' \uplus F_2' \uplus \Delta$, where Δ is a multiset of pairs of complementary conflicting updates.

The multiset F_1 is clearly a superset of F_1' since the latter is obtained by executing rules when r_2 is deleted from the initial view. Similarly, F_2 is a superset of F_2' since the former is obtained by executing rules when r_1 is inserted into the view.

Let $F_1 = F_1' \uplus \Delta_1$ and $F_2 = F_2' \uplus \Delta_2$. We need to show that $\Delta_1 \uplus \Delta_2$ contains a multiset of pairs of complementary conflicting updates. More specifically, we can show that for any insertion updates in $u \in \Delta_1$ there its complementary updates $\bar{u} \in \Delta_2$.

Updates that are in Δ_1 are generated by firing delta-rules that contain $\langle \text{INS}, r_1\rangle$ and either r_2 or r_2^ν in the body. Updates that are in Δ_2 are generated by firing delta-rules that contain $\langle \text{DEL}, r_2\rangle$ and either r_1 or r_1^ν in the body. Next we show that there is one-to-one mapping between the delta-rules that generate an update u in Δ_1 and the delta-rules that generate an update $\bar{u}$ in Δ_2.

Any insertion update u in Δ_1 is necessarily fired by rules of the following two forms:

u :- $\cdots, r_2^\nu, \cdots, \Delta r_1, \cdots$, which we call a_1
and u :- $\cdots, \Delta r_1, \cdots r_2 \cdots$, which we call a_2.

Any deletion update u in Δ_2 is necessarily fired by rules of the following two forms:

u :- $\cdots, r_1^\nu, \cdots, \Delta r_2, \cdots$, which we call b_1
and u :- $\cdots, \Delta r_2, \cdots r_1 \cdots$, which we call b_2.

Notice that there is a one-to-one mapping between a_1 and b_2, and a one-to-one mapping between a_2 and b_1. In other words, in the first execution sequence, a_1 is fired when $\langle \text{INS}, r_1\rangle$ is picked, and b_2 is fired when $\langle \text{DEL}, r_2\rangle$ is picked. Furthermore, a_1 and b_2 generates a pair of complementary conflicting updates, and so do a_2 and b_1.

Therefore, $F_1 \uplus F_2 = F_1' \uplus F_2' \uplus \Delta_1 \uplus \Delta_2$, and $\Delta_1 \uplus \Delta_2$ contains pairs of complementary conflicting updates. □

Proof of Lemma 16

PROOF. We only show the case when u is an insertion, and the second case can be proved similarly. Let $u = \langle \text{INS}, p\rangle$. By examining the two execution sequences, we know that

$F_1 = \biguplus_{u_0 \in \mathcal{H} \uplus \{u\}} firRules(u_0, \mathcal{K} \uplus \mathcal{H}_I^{t\nu} \uplus \{p^\nu\} \setminus \mathcal{H}_D^{t\nu}, \mathcal{R})$,
$F_1' = \biguplus_{u_0 \in \mathcal{H}} firRules(u_0, \mathcal{K} \uplus \mathcal{H}_I^{t\nu} \setminus \mathcal{H}_D^{t\nu}, \mathcal{R})$,
$F_2' = firRules(u, \mathcal{K} \uplus \mathcal{H}_I^{t\nu} \uplus \mathcal{H}_I^{t} \uplus \{p^\nu\} \setminus \mathcal{H}_D^{t\nu} \uplus \mathcal{H}_D^{t}), \mathcal{R})$,
$F_2 = F_1' \uplus F_2'$

where we write $\mathcal{H}_I^{t\nu}$ ($\mathcal{H}_D^{t\nu}$ respectively) to denote the multiset that contains p^ν if and only if $\langle \text{INS}, p\rangle$ ($\langle \text{DEL}, p\rangle$ respectively) is in $\mathcal{H}$. We write $\mathcal{H}_I^{t}$ ($\mathcal{H}_D^{t}$ respectively) to denote the multiset that contains p if and only if $\langle \text{INS}, p\rangle$ ($\langle \text{DEL}, p\rangle$ respectively) is in $\mathcal{H}$.

Let's further rewrite F_1 to be $F_1'' \uplus F_2''$
where $F_1'' = \biguplus_{u_0 \in \mathcal{H}} firRules(u_0, \mathcal{K} \uplus \mathcal{H}_I^{t\nu} \uplus \{p^\nu\} \setminus \mathcal{H}_D^{t\nu}, \mathcal{R})$, and $F_2'' = firRules(u, \mathcal{K} \uplus \mathcal{H}_I^{t\nu} \uplus \{p^\nu\} \setminus \mathcal{H}_D^{t\nu}, \mathcal{R})$.

F_1'' is a superset of F_1'. Let $F_1'' = F_1' \uplus \Delta_I \uplus \Delta_D$.

Any update $\langle \text{INS}, r_1\rangle \in \Delta_I$ is generated by a delta-rule that contains p^ν and an insertion update $\langle \text{INS}, q\rangle \in \mathcal{H}$ in the body:
$\langle \text{INS}, r_1\rangle$:- $\cdots, p^\nu, \cdots, \langle \text{INS}, q\rangle, \cdots$, which we call a_1.

Any update $\langle \text{DEL}, r_1'\rangle \in \Delta_D$ is generated by a delta-rule that contains p^ν and a deletion update $\langle \text{DEL}, q\rangle \in \mathcal{H}$ in the body:
$\langle \text{DEL}, r_1'\rangle$:- $\cdots, p^\nu, \cdots, \langle \text{DEL}, q\rangle, \cdots$, which we call a_2.

The relation between F_2'' and F_2' is more complicated. What we can show is the following $F_2'' \uplus \Delta_I' = F_2' \uplus \Delta_I''$ where $\Delta_I' = \Delta_I$, and Δ_I'' contains all the complimentary updates to the ones in Δ_D, nothing else.

We would like to show that there is a one-to-one mapping between the delta-rules that are fired to generate Δ_I in the bigger complete iteration (the first execution sequence), and the delta-rules that are fired to generate Δ_I' in the PSN iteration (the second part of the second execution sequence).

The only updates that are in F_2', but not in F_2'' are due to $\mathcal{H}_I^t$. Therefore, all insertion updates in Δ_I' are generated by firing delta-rules that contain u and q, where $\langle \text{INS}, q\rangle \in \mathcal{H}$, in the body:
$\langle \text{INS}, r_1\rangle$:- $\cdots, u, \cdots, q, \cdots$, which we call b_1.

By the definition of delta-rules, there is one-to-one mapping between a_1 and b_1. Consequently, $\Delta_I = \Delta_I'$.

We also need to show that there is a one-to-one mapping between the delta-rules that are fired to generate Δ_I'', and the delta-rules that are fired to generate Δ_D.

The only updates that are in F_2'', but not in F_2' are due to $\mathcal{H}_D^t$, which is deleted from the view before the PSN iteration. Therefore, all insertion updates in Δ_I'' are generated by firing delta-rules that contain u and q, where $\langle \text{DEL}, q\rangle \in \mathcal{H}$, in the body:
$\langle \text{INS}, r_1\rangle$:- $\cdots, u, \cdots, q, \cdots$, which we call b_2.

By the definition of delta-rules, there is one-to-one mapping between a_2 and b_2. Consequently, Δ_I'' contains all the complementary updates to those ones that are in Δ_D, which we denote by $\bar{\Delta}_D$.

Finally, we obtain the following: $F_1'' = F_1' \uplus (\Delta_I \uplus \Delta_D)$ and $F_2'' \uplus \Delta_I = F_2' \uplus \bar{\Delta}_D$. We know the following by union both sides of the above equations: $F_1'' \uplus F_2'' \uplus \Delta_I = F_1' \uplus (\Delta_I \uplus \Delta_D) \uplus F_2' \uplus \bar{\Delta}_D$. We can conclude that $F_1 = F_2 \uplus \Delta_D \uplus \bar{\Delta}_D$. Therefore, F_1 and F_2 only differs in pairs of complementary conflicting updates. □

Proof of Lemma 17.

PROOF. Assume that $u_c^I = \langle \text{INS}, p\rangle$ and $u_c^D = \langle \text{DEL}, p\rangle$

We first show that for any insertion update, u, created by firing delta-rules that contains $\langle \text{INS}, p\rangle$ in the body, there is exactly one deletion update $\bar{u}$ that is created at an iteration no later than the one where u_c^D is picked.

Let's assume that u is created by firing the following delta-rule:
u :- $b_1, \ldots, b_n, \langle \text{INS}, p\rangle, b_{n+1}, \ldots, b_{n+m}$.

The update $\bar{u}$ can be created either by a deletion update for b_i which is picked before u_c^D; or by the time u_c^D is processed none of the predicates (b_i) in the body has been deleted, in which case $\bar{u}$ will be generated by firing the following delta-rule.
$\bar{u}$:- $b_1, \ldots, b_n, \langle \text{DEL}, p\rangle, b_{n+1}, \ldots, b_{n+m}$.

This means that only pairs of complementary conflicting updates are propagated by the insertion and deletion of p. Using the same reasoning above, these pairs of conflicting updates created will also cause the propagation of conflicting pairs of updates only. For the rest of the proof, we call all these updates as *p-propagated updates*.

Then, in this subexecution, we use Lemma 15 to permute deletion updates to the right of insertion updates eagerly. In the process, new conflicting updates are generated, which will be dealt later. Finally, we use Lemma 14 to permute insertion updates (respectively, deletion updates), so that the propagated updates are picked last and in the same order. That is, if the propagated insertion update u_1 is picked before the propagated insertion update u_2, then the deletion update $\bar{u_1}$ is picked before $\bar{u_2}$.

Next, we define ID executions. A PSN execution is an ID execution if it has the following form:

$s_0 \xRightarrow{psn} (\mathcal{U}_I)s_1 \xRightarrow{psn} (\mathcal{U}_P)s_2 \xRightarrow{psn} (\mathcal{U}_D)s_3 \xRightarrow{psn} (\mathcal{U}'_P)s_4$,

where for all $u \in \mathcal{U}_I$, u is a non-p-propagated insertion update, for all $u \in \mathcal{U}_P$, u is a p-propagated insertion update, and for all $u \in \mathcal{U}_D$, u is a non-p-propagated deletion update, and for all $u \in \mathcal{U}'_P$, u is a p-propagated deletion update. Furthermore, for all $u \in \mathcal{U}_P$ then $\bar{u} \in \mathcal{U}_2$ and vice-versa. We denote an ID execution as $s \xRightarrow{ID} s'$.

We show that any PSN execution can be transformed into a sequence of two consecutive ID executions. The first ID execution is formed by using repeatedly using Lemma 15 to permute deletion updates to the right of insertion updates. In the process, new conflicting updates are generated, which will be used to form the second ID execution. In the end, we obtain a PSN-execution where all insertion updates are picked before deletion updates. Now we use Lemma 14 to permute insertion updates (respectively, deletion updates), so that the p-propagated updates are picked after all the non p-propagated updates are picked. This is possible because by its definition, non p-propagated updates cannot be generated by firing a delta-rule that uses p-propagated updates. Now we have obtained our first ID execution. This is not a complete PSN run because in the first step, we have generated new pairs of complementary conflicting updates.

Next, we construct the second ID execution by complete the execution of the program. We eagerly pick non-p-propagated insertion updates until only none is left, then we pick all p-propagated insertion updates. After that, we pick non-p-propagated deletion updates; then, we finish by picking all p-propagated deletion updates.

Now we have obtained a complete run of PSN, of the following form: $\langle \mathcal{K}_1, \mathcal{U}_1, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_2, \mathcal{U}_2, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_3, \emptyset, \emptyset, \emptyset \rangle$, where the view in $\mathcal{K}_2$ is the same as the original PSN execution, which is guaranteed by Lemma 15 and Lemma 14.

Next we show that we can prune an ID execution to contain only non-p-propagated updates without changing the final view.

Given an ID execution,

$$\begin{aligned}
&\langle \mathcal{K}, \mathcal{U}, \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}_I)\langle \mathcal{K} \uplus \mathcal{U}_I^t, \mathcal{U} \setminus \mathcal{U}_I \uplus F_I, \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}_P)\langle \mathcal{K} \uplus \mathcal{U}_I^t \uplus \mathcal{U}_P^t, \mathcal{U} \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_P \uplus F_P, \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}_D)\langle \mathcal{K} \uplus \mathcal{U}_I^t \uplus \mathcal{U}_P^t \setminus \mathcal{U}_D^t, \\
&\qquad \mathcal{U} \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_P \uplus F_P \setminus \mathcal{U}_D \uplus F_D, \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}'_P)\langle \mathcal{K} \uplus \mathcal{U}_I^t \uplus \mathcal{U}_P^t \setminus \mathcal{U}_D^t \setminus \mathcal{U}_P'^t, \\
&\qquad \mathcal{U} \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_P \uplus F_P \setminus \mathcal{U}_D \uplus F_D \setminus \mathcal{U}'_P \uplus F'_P, \emptyset, \emptyset \rangle
\end{aligned}$$

Let $\mathcal{U}'$ contain all the non-p-propagated updates in $\mathcal{U}$, and we generate a PSN execution that only pick non-p-propagated updates as follows.

$$\begin{aligned}
&\langle \mathcal{K}, \mathcal{U}', \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}_I)\langle \mathcal{K} \uplus \mathcal{U}_I^t, \mathcal{U}' \setminus \mathcal{U}_I \uplus F_I, \emptyset, \emptyset \rangle \\
&\xRightarrow{psn} (\mathcal{U}_D)\langle \mathcal{K} \uplus \mathcal{U}_I^t \setminus \mathcal{U}_D^t, \mathcal{U}' \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_D \uplus F'_D, \emptyset, \emptyset \rangle
\end{aligned}$$

Compared with the original ID execution, we have the following invariants.

First, $\mathcal{K} \uplus \mathcal{U}_I^t \uplus \mathcal{U}_P^t \setminus \mathcal{U}_D^t \setminus \mathcal{U}_P'^t = \mathcal{K} \uplus \mathcal{U}_I^t \setminus \mathcal{U}_D^t$ because $\mathcal{U}'_P$ contains the complement of $\mathcal{U}_P$.

Second, $\mathcal{U}' \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_D \uplus F'_D$ contains only the non-p-propagated updates in $\mathcal{U} \setminus \mathcal{U}_I \uplus F_I \setminus \mathcal{U}_P \uplus F_P \setminus \mathcal{U}_D \uplus F_D \setminus \mathcal{U}'_P \uplus F'_P$. This is because the only updates that contain non-p-propagated updates are $\mathcal{U}'$, F_I and F'_D; and $F_D \supseteq F'_D$.

We perform the above rewriting separately to both ID executions in $\langle \mathcal{K}_1, \mathcal{U}_1, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_2, \mathcal{U}_2, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_3, \emptyset, \emptyset, \emptyset \rangle$.

We obtain the following: $\langle \mathcal{K}_1, \mathcal{U}'_1, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_2, \mathcal{U}'_2, \emptyset, \emptyset \rangle$ and $\langle \mathcal{K}_2, \mathcal{U}''_2, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_3, \emptyset, \emptyset, \emptyset \rangle$.

The invariants tell us that $\mathcal{U}'_1$ contains all non-p-propagated updates in $\mathcal{U}_1$ and nothing else, and both $\mathcal{U}'_2$ and $\mathcal{U}''_2$ contains all the non-p-propagated updates in $\mathcal{U}_2$ and nothing else. Therefore, we know that $\mathcal{U}_1 = \mathcal{U}'_1 \uplus \{\langle \text{INS}, p \rangle, \langle \text{DEL}, p \rangle\}$, and $\mathcal{U}'_2 = \mathcal{U}''_2$. Finally, we obtain the valid PSN execution sequence: $\langle \mathcal{K}_1, \mathcal{U}'_1, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_2, \mathcal{U}'_2, \emptyset, \emptyset \rangle \xRightarrow{ID} \langle \mathcal{K}_3, \emptyset, \emptyset, \emptyset \rangle$. □

B. PROBLEMS OF PREVIOUS WORK IN AN ASYNCHRONOUS SETTING

The main problem with the algorithm in [10] is that they do not store enough information in the annotation of tuples to be able to differentiate between when an update is due to a cyclic proof and when an update arrived out-of-order. In particular, they annotate tuples with provenance polynomials [5] constructed using only the base tuples used to derive tuple and not the intermediate derived tuples. For instance, consider the same *DDlog* program used in Section 5 to illustrate that our PSN algorithm that detects cycles terminates:

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

The view in their setting for this program when `a` is true is `(a,{a})`, `(p,{a})`, `(q,{a})` where we elide the `(@X)` symbols. All tuples are derived by only using the base tuple `a` and therefore their annotations consist only of the monomial `a`. Clearly, using annotations containing just base tuples is not enough to detect cycles in derivations. For instance, an update inserting `(p,{a})` could be derived due to the a derivation with no cycles or due to a cyclic derivation obtained by using the last two rules of program.

In order to avoid divergence, one would need to discard the latter type of updates, as in our PSN algorithm. They are able to detect such updates but only when one assumes that all transmission channels are FIFO, that is, when messages are not reordered and guarantee termination by discarding updates. To illustrate how their algorithm works, consider again the program above and the same view. Assume that there is a deletion of `a`, that is, the existence of the deletion update $\langle$DEL, `(a,{a})`$\rangle$. When this update is processed, node `1` creates $\langle$DEL, `(p,{a})`$\rangle$, which on the other hand is processed by node `2`, creating the update $\langle$DEL, `(q,{a})`$\rangle$. Finally, node `2` processes the latter, creating again the deletion update $\langle$DEL, `(p,{a})`$\rangle$. When this update is received by node 1, the tuple `(p,{a})` is not in the view, since 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.

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: $\langle$DEL, `(a,{a})`$\rangle$ and $\langle$INS, `(a,{a})`$\rangle$. If the deletion update is processed first by node `0`, it will be discarded since the tuple `(a,{a})` is not present in its view. The insertion update on the other hand would be processed, generating eventually new insertion updates for all the tuples in the program. Hence, the final view obtained by their algorithm is `(a,{a})`, `(p,{a})`, `(q,{a})`, whereas the correct view is the empty set.