Declarative Networking: Language, Execution and Optimization

Boon Thau Loo[*] Tyson Condie[*] Minos Garofalakis[†] David E. Gay[†] Joseph M. Hellerstein[*]
Petros Maniatis[†] Raghu Ramakrishnan[‡] Timothy Roscoe[†] Ion Stoica[*]
[*]UC Berkeley, [†]Intel Research Berkeley and [‡]University of Wisconsin-Madison

ABSTRACT

The networking and distributed systems communities have recently explored a variety of new network architectures, both for application-level overlay networks, and as prototypes for a next-generation Internet architecture. In this context, we have investigated *declarative networking*: the use of a distributed recursive query engine as a powerful vehicle for accelerating innovation in network architectures [23, 24, 33]. Declarative networking represents a significant new application area for database research on recursive query processing. In this paper, we address fundamental database issues in this domain. First, we motivate and formally define the Network Datalog (*NDlog*) language for declarative network specifications. Second, we introduce and prove correct relaxed versions of the traditional semi-naïve query evaluation technique, to overcome fundamental problems of the traditional technique in an asynchronous distributed setting. Third, we consider the dynamics of network state, and formalize the "eventual consistency" of our programs even when bursts of updates can arrive in the midst of query execution. Fourth, we present a number of query optimization opportunities that arise in the declarative networking context, including applications of traditional techniques as well as new optimizations. Last, we present evaluation results of the above ideas implemented in our P2 declarative networking system, running on 100 machines over the Emulab network testbed.

1. INTRODUCTION

The database literature has a rich tradition of research on recursive query languages and processing. This work has influenced commercial database systems to a certain extent. However, recursion is still considered an esoteric feature by most practitioners, and research in the area has had limited practical impact. Even within the database research community, there is longstanding controversy over the practical relevance of recursive queries, going back at least to the Laguna Beach Report [7], and continuing into relatively recent textbooks [35].

In more recent work, we have made the case that recursive query technology has a natural application in the design of Internet infrastructure. We presented an approach called *declarative networking* that enables declarative specification and deployment of distributed protocols and algorithms via distributed recursive queries over network graphs [23, 24, 33]. We recently described how we implemented and deployed this concept in a system called *P2* [23, 33]. Our high-level goal is to provide a software environment that can accelerate the process of specifying, implementing, experimenting with and evolving designs for network architectures.

Declarative networking is part of a larger effort to revisit the current Internet Architecture, which is considered by many researchers to be fundamentally ill-suited to handle today's network uses and abuses [13]. While radical new architectures are being proposed for a "clean slate" design, there are also many efforts to develop application-level "overlay" networks on top of the current Internet, to prototype and roll out new network services in an evolutionary fashion [26]. Whether one is a proponent of revolution or evolution in this context, there is agreement that we are entering a period of significant flux in network services, protocols and architectures.

In such an environment, innovation can be better focused and accelerated by having the right software tools at hand. Declarative query approaches appear to be one of the most promising avenues for dealing with the complexity of prototyping, deploying and evolving new network architectures. The forwarding tables in network routing nodes can be regarded as a view over changing ground state (network links, nodes, load, operator policies, etc.), and this view is kept correct by the maintenance of distributed queries over this state. These queries are necessarily recursive, maintaining facts about arbitrarily long multi-hop paths over a network of single-hop links.

Our initial forays into declarative networking have been promising. First, in *declarative routing* [24], we demonstrated that recursive queries can be used to express a variety of well-known wired and wireless routing protocols in a compact and clean fashion, typically in a handful of lines of program code. We also showed that the declarative approach can expose fundamental connections: for example, the query specifications for two well-known protocols – one for wired networks and one for wireless – differ only in the order of two predicates in a single rule body. Moreover, higher-level routing concepts (*e.g.*, QoS constraints) can be achieved via simple modifications to the queries. Second, in *declarative overlays* [23], we extended our framework to support more complex application-level overlay networks such as multicast overlays and distributed hash tables (DHTs). We demonstrated a working implementation of the Chord [34] overlay lookup network specified in 47 Datalog-like rules, versus *thousands* of lines of C++ for the original version.

Our declarative approach to networking promises not only flexibility and compactness of specification, but also the potential to statically check network protocols for security and correctness properties [11]. In addition, dynamic runtime checks to test distributed properties of the network can easily be expressed as declarative queries, providing a uniform framework for network specification, monitoring and debugging [33].

[*]UC Berkeley authors funded by NSF grants 0205647, 0209108, and 0225660, and a gift from Microsoft.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

1.1 The Database Research Agenda

In our earlier declarative networking proposals, we focused primarily on addressing problems in networking and distributed systems. In doing so, we set aside important and challenging questions of language semantics, distributed execution strategies, and correctness under network dynamics, all of which are essential for the practical realization of declarative networks.

In this paper, we explore several of these research issues from the database perspective. We implemented our ideas in the P2 system, and present evaluations of many of our optimizations in realistic large-scale distributed experiments. Specifically, the main contributions of this paper are as follows:

- We motivate and formally define the *NDlog* language for declarative network specification. *NDlog* is a subset of Datalog that makes explicit the link graph of the network and the partitioning of data across nodes. As part of *NDlog*, we introduce the concept of *link-restricted* rules, which guarantees that all rules can be rewritten to be executed locally at individual nodes, and all communication for each rewritten rule only involves sending messages along links (Section 2).
- We introduce and prove correct relaxed versions of the semi-naïve execution strategy called *buffered semi-naïve* and *pipelined semi-naïve* evaluation. These techniques overcome fundamental problems of semi-naïve evaluation in an asynchronous distributed setting, and should be of independent interest outside the context of declarative networking: they significantly increase the flexibility of semi-naïve evaluation to order the derivation of facts (Section 3).
- In the declarative network setting, transactional isolation of updates from concurrent queries is not useful; network protocols must incorporate concurrent updates about the state of the network while they run. We address this by formalizing the typical distributed systems notion of "eventual consistency" in our context of derived data. Using techniques from materialized recursive view maintenance, we incorporate updates to base facts *during* query execution, and still ensure well-defined eventual consistency semantics. This is of independent interest beyond the network setting when handling updates and long-running recursive queries (Section 4).
- We present a number of query optimization opportunities that arise in the declarative networking context, including applications of traditional techniques (*e.g.*, aggregate selections and magic-sets rewriting), as well as new optimizations for work-sharing, caching, and cost-based optimizations based on graph statistics. Again, many of these ideas can be applied outside the context of declarative networking or distributed implementations (Section 5).
- We present evaluation results from a distributed deployment involving 100 machines connected by the Emulab [10] network testbed, running prototypes of our optimization techniques implemented as modifications to the P2 declarative overlay system (Section 6).

2. DATA AND QUERY MODEL

We first provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey [28]. A Datalog program consists of a set of declarative *rules* and a query. A Datalog *rule* has the form p :- $q_1, q_2, ..., q_n$., which can be read informally as "q_1 and q_2 and ... and q_n implies p". p is the *head* of the rule, and $q_1, q_2, ..., q_n$ is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* applied to *fields* (variables and constants), or function symbols applied to fields. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial. The commas separating the predicates in a rule are logical conjuncts (*AND*); the order in which predicates appear in a rule body also has no semantic significance (though implementations typically employ a left-to-right execution strategy). The query specifies the output of interest.

The predicates in the body and head of Datalog rules are relations, and we will refer to them interchangeably as predicates, relations, or tables. Each relation has a *primary key*, which consists of a set of fields that uniquely identifies each tuple within the relation. We allow the primary key to be specified for stored ("extensional") relations; in the absence of other information, the primary key is the full set of attributes in the relation.

The names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Most implementations of Datalog enhance it with a limited set of function calls (which start with "*f_*" in our syntax), including boolean predicates, arithmetic computations and simple list manipulation (*e.g.*, the *f_concatPath* function in our first example). Aggregate constructs are represented as functions with field variables within angle brackets (<>). For most of our discussion, we will not consider negated predicates; we will return to the topic of negation as part of our future work (Section 8).

As an example, the following program computes the shortest paths between all pairs of nodes in a graph. The program has four rules (which for convenience we label R1-R4), and takes as input a stored ("extensional") relation $link(src, dst, cost)$. R1 and R2 are used to derive "paths" in the graph, represented as tuples in the derived ("intensional") relation $path(src, dst, nextHop, pathVector, \ldots)$. The *src* and *dst* fields represent the endpoints of the path; the *pathVector* is a string encoding the full path. We discuss *nextHop* later. Given the *path* relation, Rule R4 computes the shortest paths as the derived relation $shortestPath(src, dst, pathVector, cost)$. R3 derives the relation $spCost(src, dst, mincost)$ that computes the minimum cost for each (src,dst) group for all input paths. The rule *Query* specifies shortestPath tuples as the result tuples. R2 is a *linear* rule, since there is only one recursive literal in the body. Rules with more than one recursive literal in the body are *non-linear*.

R1: path(S,D,D,P,C) :- link(S,D,C), P = *f_concatPath*(link(S,D,C), nil).
R2: path(S,D,Z,P,C) :- link(S,Z,C_1), path(Z,D,Z_2,P_2,C_2),
    C = C_1 + C_2, P = *f_concatPath*(link(S,Z,C_1),P_2).
R3: spCost(S,D,min<C>) :- path(S,D,Z,P,C).
R4: shortestPath(S,D,P,C) :- spCost(S,D,C), path(S,D,Z,P,C).
Query: shortestPath(S,D,P,C).

Rule R1 produces one-hop paths from existing link tuples, and Rule R2 recursively produces path tuples of increasing cost by matching the destination fields of existing links to the source fields of previously computed paths. The matching is expressed using the repeated "Z" variable in $link(S, Z, C_1)$ and $path(Z, D, Z_2, P_2, C_2)$ of rule R2. Intuitively, rule R2 says that if there is a link from node *S* to node *Z*, and there is a path from node *Z* to node *D*, then there is a path from node *S* to node *D* via *Z*. In the presence of path cycles, the query never terminates, as R1 and R2 will generate paths of ever increasing lengths. However, this can be fixed with a well-known query rewrite (Section 5.1.1) when costs are positive.

2.1 Network Datalog

In this section, we introduce the data and query model that we propose for declarative networking. The language we present is *Network Datalog* (*NDlog*), a restricted variant of traditional Datalog intended to be computed in distributed fashion on physical network graphs. In describing our model, we use the *NDlog* query shown in Figure 1, which performs distributed computation of shortest paths.

One of the novelties of our setting, from a database perspective, is that data is distributed and relations may be partitioned across sites. To ease the generation of efficient query plans in such a system, *NDlog* gives the query writer *explicit* control on data placement and movement. Specifically, *NDlog* uses a special data type,

SP1: path(**@S**,@D,@D,P,C) :- #link (**@S**,@D,C),
P = *f_concatPath*(link(**@S**,@D,C), nil).
SP2: path(**@S**,@D,@Z,P,C) :- #link (**@S**,@Z,C_1),
path(**@Z**,@D,@Z_2,P_2,C_2), C = C_1 + C_2,
P = *f_concatPath*(link(**@S**,@Z,C_1),P_2).
SP3: spCost(**@S**,@D,min<C>) :- path(**@S**,@D,@Z,P,C).
SP4: shortestPath(**@S**,@D,P,C) :- spCost(**@S**,@D,C),
path(**@S**,@D,@Z,P,C).
Query: shortestPath(**@S**,@D,P,C).

Figure 1: ***Shortest-Path Query in* NDlog .**

address, to specify a network location. Names of address variables and constants are prepended with "@". More formally, we have the following definition:

Definition 1 A *location specifier* is an attribute of type address in a predicate that indicates the network storage location of each tuple.

As a matter of notation, we require the location specifier to be the first field in all predicates, and we highlight it in **bold** for clarity. For example, the location specifier of *link*(**@S**,@D,C) is **@S**.

Another novelty of our setting is that we assume a network graph that is not fully connected, *i.e.*, a node can communicate *directly* with only a subset of nodes in the system. This allows us to model the physical connectivity of a typical autonomous system in the Internet, where each node is connected to relatively few other nodes. In contrast, both traditional parallel query processors and more recent distributed query engines, such as PIER [17], assume a fully connected network graph, where messages can be sent directly from any node to any other node in the system. Parallel systems achieve this by engineering (and provisioning) the interconnection network, while PIER uses overlay routing to connect any two nodes.

To express the constraint that a node can send data only to another node with which it is physically connected, we introduce the concept of *link relation*, which is defined as follows:

Definition 2 A *link relation* is a stored ("extensional") relation $link(@src, @dst, ...)$ representing the connectivity information of the network being queried.

The first two fields of each link table entry contain the source and destination addresses of a network link respectively, followed by an arbitrary number of other fields (typically metrics) describing the link. In this paper, we constrain all links to be bidirectional, *i.e.*, if there is a network edge from a node to its neighbor, the reverse must be true[1]. In all our example queries, we utilize only one link table. In practice, there can be multiple such tables used by different rules.

Given that we will be executing queries across network links, it is useful to identify queries that do not require communication:

Definition 3 *Local rules* are rules that have the same location specifier in each predicate, including the head.

Local rules can be executed without any distributed logic. Rules SP1, SP3 and SP4 are local. SP2 is a non-local rule since the *link* and *path* body predicates are stored at different locations.

In *NDlog*, the evaluation of a rule must depend only on communication along the physical links. To this end, we introduce the following:

Definition 4 A *link literal* is a link relation that appears in the body of a rule prepended with the "#" symbol.

Given the preceding definitions, we are ready to define a simple syntactic constraint on the rules to ensure that communication takes place only along the physical links:

Definition 5 A *link-restricted* rule is either a local rule, or a rule with the following properties:

- There is exactly one link literal in the body
- All other literals (including the head predicate) have their location specifier set to either the first (source) or second (destination) field of the link literal.

This syntactic constraint precisely captures the requirement that we be able to operate directly on a network whose link connectivity is not a full mesh. Further, as we demonstrate in Section 3, link-restriction also guarantees that all programs with only link-restricted rules can be rewritten into a canonical form where every rule body can be evaluated on a single node. In addition, all communication for each rewritten rule only involves sending messages along links. The following is an example of a link-restricted rule:

p(**@D**,...) :- #link (**@S**,@D,...),p_1(**@S**,...),p_2(**@S**,...), ..., p_n(**@S**,...).

The rule body of this example is executed at @S and the resulting *p* tuples are sent to @D, preserving the communication constraints along links. Note that this example's body predicates all have the same location specifier: @S, the source of the link. In contrast, rule SP2 of Figure 1 is link-restricted but has some relations whose location specifier is the source, and others whose location specifier is the destination; this needs to be rewritten as described in Section 3.

Given these preliminaries, we are now ready to present our language *NDlog*:

Definition 6 A Network Datalog (*NDlog*) program is a Datalog program that satisfies the following syntactic constraints:

1. **Location specificity:** Each predicate has a location specifier as its first attribute
2. **Address type safety:** A variable that appears once in a rule as an address type must not appear elsewhere in the rule as a non-address type.
3. **Stored link relations:** Link relations never appear in the head of a rule with a non-empty body (*i.e.*, they are stored, not derived).
4. **Link-restriction:** Any non-local rules in the program are link-restricted by some link relation.

Since *NDlog* is a subset of Datalog, the semantics of a valid *NDlog* program are exactly those of Datalog.

2.2 Shortest Path Example

To illustrate *NDlog*, we step through an execution of the *shortest-path* query above to illustrate derivation and communication of tuples as the query is computed. We make use of the example network in Figure 2. Our discussion is necessarily informal since we have not yet presented our distributed implementation strategies; in the next section, we show in greater detail the steps required to generate the execution plan. Here, we focus on a high-level understanding of the data movement in the network during query processing.

We will describe communication in *iterations*, where at each iteration, each network node generates *paths* of increasing hop count, and then propagates these paths to neighbor nodes along links. In the 1st iteration, all nodes initialize their local path tables to 1-hop paths using SP1. In the 2nd iteration, using SP2, each node takes the input paths generated in the previous iteration, and computes 2-hop paths, which are then propagated to its neighbors. For example, $path(\mathbf{a}, d, b, [a, b, d], 6)$ is generated at node b using $path(b, d, d, [b, d], 1)$ from the 1st iteration, and propagated to node a. In addition to storing the entire path vector, each *path* tuple also contains the *nextHop* attribute, which indicates for each path the next hop to route the message in the network. In fact, many network protocols propagate only the *nextHop* and avoid sending the entire path vector.

[1]In practice, some networks may not have symmetric links. Our framework can be extended to handle this, but generalizing the discussion in that manner complicates our presentation and is out of the scope of this paper.

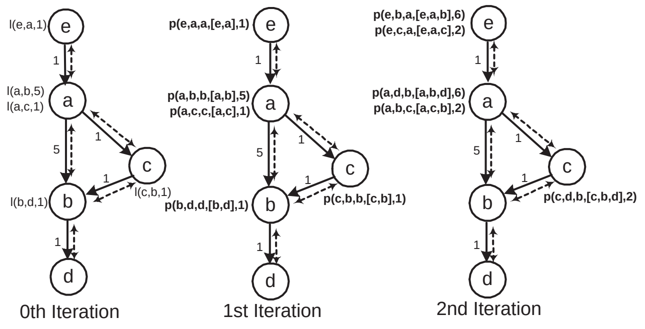

Figure 2: ***Nodes in the network are running the shortest-path query. We only show newly derived tuples at each iteration. For simplicity, we show only the derived paths along the solid lines even though the network connectivity is bidirectional (dashed lines).***

As paths are being computed, the shortest paths are also incrementally computed. For example, node a computes $path(\mathbf{a},b,b,[a,b],5)$ using rule SP1, and then sets its shortest path to $shortestPath(\mathbf{a},b,[a,b],5)$ using rule SP4. In the next iteration, node a receives $path(\mathbf{a},b,c,[a,c,b],2)$ from node c which has lower cost compared to the previous shortest cost of 5, and hence a new $shortestPath(\mathbf{a},b,[a,b],2)$ replaces the previous value.

2.3 Expressiveness

In previous work [24] we argued that executing a shortest path distributed Datalog query closely resembles the distributed computation of the well-known path vector [25] protocol. In proposals for declarative networks [23, 24], Datalog-like programs were used for a variety of networking tasks, including standard routing protocols such as *distance vector* [25] and *dynamic source routing* [20], and more complex networks such as multicast trees and the Chord network [34]. We note that *NDlog* is flexible enough for expressing most of these programs efficiently, and provides the advantages of having clear semantics as described above (something that is not available in the original language for P2 described in [23]) and a clearly defined link-restricted implementation as described below.

3. EXECUTION PLAN GENERATION

Having illustrated the intended execution of an example program, we now describe the steps required to automatically generate an execution plan from a *NDlog* program. We first focus on generating an execution plan in a centralized implementation, before extending the techniques to the network scenario.

3.1 Centralized Plan Generation

In generating the centralized plan, we utilize the well-known *semi-naïve fixpoint* [3,4] evaluation mechanism that ensures no redundant evaluations. As a quick review, in semi-naïve (SN) evaluation, input tuples computed in the previous iteration of a recursive rule execution are used as input in the current iteration to compute new tuples. Any new tuples that are generated for the first time in the current iteration are then used as input to the next iteration. This is repeated until a fixpoint is achieved (*i.e.*, no new tuples are produced).

The semi-naïve rewritten rule for rule SP2 is shown below:

SP2-1: $\triangle$path^{new}(**@S**,@D,@Z,P,C) :- `#link` (**@S**,@Z,C$_1$),
$\triangle$path^{old}(**@Z**,@D,@Z$_2$,P$_2$,C$_2$), C = C$_1$ + C$_2$,
P = *f_concatPath*(link(**@S**,@Z,C$_1$),P$_2$).

Figure 3 shows the dataflow realization for rule SP2-1 using the conventions of P2. We will briefly explain how the semi-naïve evaluation is achieved here. Each semi-naïve rule is implemented as a *rule strand*. Each strand consists of a number of relational operators. The example strand receives new $\triangle path^{old}$ tuples generated in the previous iteration to generate new paths ($\triangle path^{new}$) which are then inserted into the *path* table (with duplicate elimination) for further processing in the next iteration.

In Algorithm 1, we show the pseudocode for a centralized P2 implementation of multiple semi-naïve rule strands where each rule has the form $\triangle p_j^{new}$:- p_1^{old},..., p_{k-1}^{old},$\triangle p_k^{old}$,p_{k+1},...,p_n,$b_1,b_2,\ldots,b_m$;[2] $p_1,\ldots,p_n$ are recursive predicates and $b_1,\ldots b_m$ are base predicates. $\triangle p_k^{old}$ refers to p_k tuples generated for the first time in the previous iteration. p_k^{old} refers to all p_k tuples generated before the previous iteration.

Algorithm 1 Semi-naïve (SN) Evaluation in P2
```
while $\exists B_k.size > 0$
    $\forall B_k$ where $B_k.size > 0, \triangle p_k^{old} \leftarrow B_k.flush()$
    execute all rule strands
    foreach recursive predicate $p_j$
        $p_j^{old} \leftarrow p_j^{old} \cup \triangle p_j^{old}$
        $B_j \leftarrow \triangle p_j^{new} - p_j^{old}$
        $p_j \leftarrow p_j^{old} \cup B_j$
        $\triangle p_j^{new} \leftarrow \emptyset$
```

In the algorithm, B_k denotes the buffer for p_k tuples generated in the previous iteration ($\triangle p_k^{old}$). Initially, p_k, p_k^{old}, $\triangle p_k^{old}$ and $\triangle p_k^{new}$ are empty. As a base case, we execute all the rules to generate the initial p_k tuples, which are inserted into the corresponding B_k buffers. Each subsequent iteration of the while loop consists of flushing all existing $\triangle p_k^{old}$ tuples from B_k and executing all rule strands to generate $\triangle p_j^{new}$ tuples, which are used to update p_j^{old}, B_j and p_j accordingly. Note that only new p_j tuples generated in the current iteration are inserted into B_j for use in the next iteration. Fixpoint is reached when all buffers are empty.

3.2 Distributed Plan Generation

In the distributed implementation of the *shortest-path* query, non-local rules whose body predicates have different location specifiers cannot be executed at a single node, since the tuples that must be joined are situated at different nodes in the network. A *rule localization* rewrite step ensures that all tuples to be joined are at the same node. This allows a rule body to be locally computable.

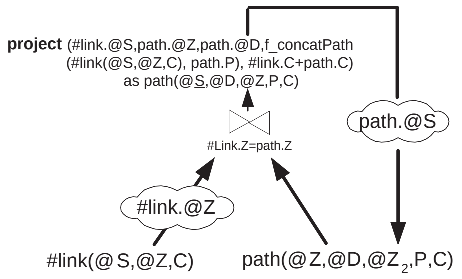
project (#link.@S,path.@Z,path.@D,f_concatPath (#link(@S,@Z,C), path.P), #link.C+path.C) as path(@S,@D,@Z,P,C)
#Link.Z=path.Z
path.@S
#link.@Z
#link(@S,@Z,C)
path(@Z,@D,@Z2,P,C)

Figure 4: ***Logical Query Plan for rule SP2 from Section 2.***

Consider rule SP2 from Section 2 where the link and path predicates have different location specifiers. These two predicates are

[2]These rules are logically equivalent to rules of the form $\triangle p_j^{new}$:- p_1,p_2,...,p_{k-1},$\triangle p_k^{old}$,p_{k+1},...,$p_n,b_1,b_2,\ldots,b_m$, and have the advantage of avoiding redundant inferences within each iteration.

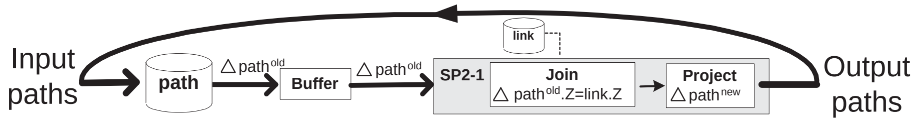

Figure 3: ***Rule strand for rule SP2-1 in P2. Output paths that are generated from the strand are "wrapped back" as input into the same strand.***

joined by a common "@Z" address field. Figure 4 shows the corresponding logical query plan depicting the distributed join. The clouds represent an "exchange"-like operator [14] that forwards tuples from one network node to another; clouds are labeled with the link attribute that determines the tuple's recipient. The first cloud (`#link`.@Z) sends link tuples to the neighbor nodes indicated by their destination address fields, to join with matching path tuples stored by their source address fields. The second cloud (*path*.*@S*) transmits for further processing new path tuples computed from the join, setting the recipient according to the source address field.

Based on the above distributed join, rule SP2 can be rewritten into the following two rules. Note that all predicates in the body of SP2a have the same location specifiers; the same is true of SP2b.

SP2a: linkD(**@Z**,@S,C) :- `#link` (**@S**,@Z,C).
SP2b: path(**@S**,@D,@Z,P,C) :- `#link` (**@Z**,@S,C_3),linkD(**@Z**,@S,C_1),
path(**@Z**,@D,@Z_2,P_2,C_2),
C = C_1 + C_2,
P = *f_concatPath*(linkD(**@Z**,@S,C_1),P_2).

The rewrite is achievable because the *link* and *path* predicates, although at different locations, share a common join address field. In Algorithm 2, we summarize the general rewrite technique for an input set of link-restricted rules R. In the pseudocode, for simplicity, we assume that the location specifiers of all the body predicates are sorted (@S followed by @D); this can be done as a preprocessing step. The algorithm as presented here assumes that all links are bidirectional, and may add a `#link` (@D,@S) to a rewritten rule to allow for backward propagation of messages.

Algorithm 2 Rule Localization Rewrite

```
proc RuleLocalization(R)
  while ∃ rule r ∈ R: h(@L,...) :- #link (@S,@D,...),
                          p_1(@S,..),..,p_i(@S,...),
                          p_{i+1}(@D,...),..,p_n(@D,..)
      R.remove(r)
      R.add(hS(@S,@D,..) :- #link (@S,@D,..),..,p_i(@S,..).)
      R.add(hD(@D,@S,..) :- hS(@S,@D,..).)
      if @L = @D
        then R.add(h(@D,..) :- hD(@D,@S,..),
                          p_{i+1}(@D,..),..,p_n(@D,..).)
        else R.add(h(@S,..) :- #link (@D,@S),hD(@D,@S..),
                          p_{i+1}(@D,..),..,p_n(@D,..).)
```

Claim 1 *Every link-restricted* NDlog *program, when rewritten using Algorithm 2, produces an equivalent program where the following holds:*

1. *The body of each rule can be evaluated at a single node.*
2. *The communication required to evaluate a rule is limited to sending derived tuples over links from a link relation.*

The equivalence statement in the above claim can be easily shown, by examining the simple factoring of each removed rule into two parts. The remainder of the claim can be verified syntactically in the added rules.

Returning to our example, after rule localization we perform the semi-naïve rewrite, and then generate the rule strands shown in Figure 5. Unlike the centralized strand in Figure 3, there are now three rule strands. The extra two strands (*SP2a@S and SP2b-2@Z*) are used as follows. Rule strand *SP2a@S* sends all existing links to the destination address field as *linkD* tuples. Rule strand *SP2b-2@Z* takes the new *linkD* tuples it received via the network and performs a join operation with the local *path* table to generate new paths.

3.3 Relaxing Semi-naïve Evaluation

In our distributed implementation, the execution of rule strands can depend on tuples arriving via the network, and can also result in new tuples being sent over the network. Traditional semi-naïve evaluation completely evaluates all rules on a given set of facts, *i.e.*, completes the *iteration,* before considering any new facts. In a distributed execution environment where messages can be delayed or lost, the completion of an iteration in the traditional sense can only be detected by a consensus computation across multiple nodes, which is expensive; further, the requirement that many nodes complete the iteration together (a "barrier synchronization" in parallel computing terminology) limits parallelism significantly by restricting the rate of progress to that of the slowest node.

We address this by making the notion of iteration local to a node. New facts might be generated through local rule execution, or might be received from another node while a local iteration is in progress. We propose and prove correct two variations of semi-naïve iteration to handle this situation: *buffered semi-naïve* (BSN) and *pipelined semi-naive* (PSN). Both approaches extend SN to work in an asynchronous distributed setting, while generating the same results as SN evaluation. We further prove that these techniques avoid duplicate inferences, which may result in generating network messages.

3.3.1 Buffered Semi-naïve

Buffered semi-naïve (BSN) is the standard SN algorithm described in Figure 1 with the following modifications: A node can start a local SN iteration at any time its local B_k buffers are non-empty. Tuples arriving over the network while an iteration is in progress are buffered for processing in the next iteration.

By relaxing the need to run an iteration to global completion, BSN relaxes SN substantially, by allowing a tuple from a traditional SN iteration to be buffered arbitrarily, and handled in some future iteration of our choice. Consequently, BSN may generate fewer tuples per iteration, but all results will eventually be generated. Since BSN uses the basic SN algorithm, the proof of correctness is straightforward and we omit it for brevity.

The flexibility offered by BSN on when to process a tuple could also be valuable outside the network setting, *e.g.*, a disk-based hash join could accumulate certain tuples across iterations, spill them to disk in value-based partitions, and process them in value batches, rather than in order of iteration number. Similar arguments for buffering apply to other query processing tricks: achieving locality in B-tree lookups, improving run-lengths in tournament sorts, etc.

3.3.2 Pipelined Semi-naïve

As an alternative to BSN, *pipelined semi-naïve* (PSN) relaxes semi-naïve evaluation to the extreme of processing each tuple as it is received. This provides opportunities for additional optimizations on a per-tuple basis, at the potential cost of set-oriented local processing. New tuples that are generated from the semi-naïve rules, as well as tuples received from other nodes, are used immediately to compute new tuples without waiting for the current (local) iteration

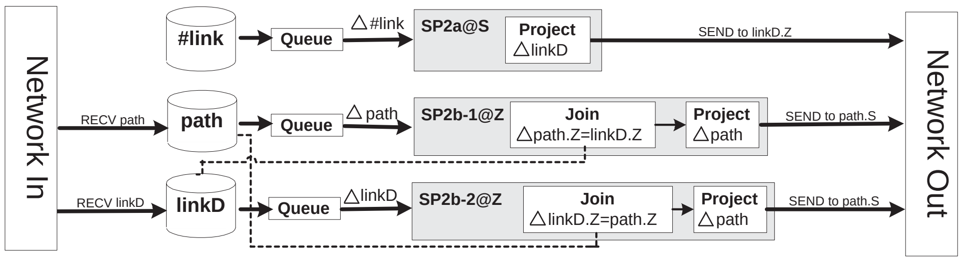

Figure 5: ***Rule strands for the distributed version of SP2 after localization in P2.***

to complete.

Algorithm 3 Pipelined Semi-naïve (PSN) Evaluation

```
while $\exists Q_k.size > 0$
    $t_k^{old,i} \leftarrow Q_k.dequeueTuple()$
    foreach rule strand execution
        $\triangle p_j^{new,i+1} :- p_1, .., p_{k-1}, t_k^{old,i}, p_{k+1}, .., p_n, b_1, b_2, ..., b_m$
        foreach $t_j^{new,i+1} \in \triangle p_j^{new,i+1}$
            if $t_j^{new,i+1} \notin p_j$
                then $p_j \leftarrow p_j \cup t_j^{new,i+1}$
                    $Q_j.enqueueTuple(t_j^{new,i+1})$
```

Algorithm 3 shows the pseudocode for PSN. Each tuple, denoted t, has a superscript (old/new, i) where i is its corresponding iteration number in SN evaluation. Each processing step in PSN consists of dequeuing a tuple $t_k^{old,i}$ from Q_k and then using it as input into all corresponding rule strands. Each resulting $t_j^{new,i+1}$ tuple is pipelined, stored in its respective p_j table (if a copy is not already there), and enqueued into Q_j for further processing. Note that in a distributed implementation, Q_j can be a queue on another node, and the node that receives the new tuple can immediately process the tuple after the enqueue into Q_j. For example, the dataflow in Figure 5 is based on a distributed implementation of PSN, where incoming *path* and *linkD* tuples received via the network are stored locally, and enqueued for processing in the corresponding rule strands.

To fully pipeline evaluation, we have also removed the distinctions between p_j^{old} and p_j in the rules. Instead, a timestamp (or monotonically increasing sequence number) is added to each tuple at arrival, and the join operator matches each tuple only with tuples that have the same or older timestamp. This allows processing of tuples immediately upon arrival, and is natural for network message handling. This represents an alternative "book-keeping" strategy to the rewriting used in SN to ensure no repeated inferences. Note that the timestamp only needs to be assigned locally, since all the rules are localized.

While PSN enables fully pipeline evaluation, it is worth noting that PSN can allow just as much buffering as BSN with the additional flexibility of full pipelining.

In Appendix A, we prove that PSN generates the same results as SN, and does not repeat any inferences. Let $FP_S(p)$ and $FP_P(p)$ denote the result set for p for using SN and PSN respectively. We show that:

Theorem 1: $FP_S(p) = FP_P(p)$

Theorem 2: *There are no repeated inferences in computing* $FP_P(p)$.

In order to compute rules with aggregation (such as SP3), we utilize incremental fixpoint evaluation techniques [27] that are amenable to pipelined query processing. These techniques can compute *monotonic aggregates* such as *min*, *max* and *count* incrementally based on the current aggregate and each new input tuple. We omit the details for lack of space.

4. SEMANTICS IN A DYNAMIC NETWORK

In practice, the state of the network is constantly changing during query execution. In contrast to transactional databases, changes to network state are not isolated from queries while they are running. Instead, as in network protocols, queries are expected to perform dynamic recomputations to reflect the most current state of the network. To better understand the semantics in a dynamic network, we consider the following two degrees of dynamism:

Continuous Update Model: In this model, we assume that updates occur very frequently – at a period that is shorter than the expected time for a typical query to reach a fixpoint. Hence, the query results never fully reflect the state of the network.

Bursty Update Model: In this idealized (but still fairly realistic model), updates are allowed to happen during query processing. However, we make the assumption that after a burst of updates, the network eventually *quiesces* (does not change) for a time long enough to allow all the queries in the system to reach a fixpoint.

In our analysis, we focus on the bursty model, since it is amenable to analysis; our results on that model provide some intuition as to the behavior in the continuous update model. Our goal in the bursty model is to achieve a variant of the typical distributed systems notion of *eventual consistency*, customized to the particulars of *NDlog*: we wish to ensure that the eventual state of the quiescent system corresponds to what would be achieved by rerunning the queries from scratch in that state. We briefly sketch the ideas here, and follow up with details in the remainder of the section.

To ensure well-defined semantics, we use techniques from materialized view maintenance [15], and consider three types of changes:

Insertion: The insertion of a new tuple at any stage of processing can be naturally handled by (pipelined) semi-naïve evaluation.

Deletion: The deletion of a base tuple leads to the deletion of any tuples that were derived from that base tuple. Deletions are carried out incrementally via (pipelined) semi-naïve evaluation by incrementally deriving all tuples that are to be deleted.

Update: An update is treated as a deletion followed by an insertion. An update to a base tuple may itself result in derivation of more updates that are propagated via (pipelined) semi-naïve evaluation.

The use of pipelined semi-naïve evaluation in the discussion can be replaced with buffered semi-naïve without changing our analysis. Since some tuples may have multiple derivations, we use the *count algorithm* [15] for keeping track of the number of derivations for each tuple, and only delete a tuple when the count is 0.

In dealing with queries with aggregates, we apply techniques for incremental computation of aggregates [27] in the presence of updates. The arrival of new tuples may invalidate existing aggregates, and incremental recomputations are cheaper than computing the entire aggregate from scratch. For example, the re-evaluation cost for min and max aggregates are shown to be $O(log\ n)$ time and $O(n)$ space for the min and max aggregates [27].

4.1 Centralized Semantics

We first provide an intuitive example for the centralized case. Figure 6 shows a *derivation tree* for $path(\mathbf{e},e,a,[e,a,b,e],7)$ based on the shortest-path query. The leaves in the tree are the *link* base tuples. The root and the intermediate nodes are tuples recursively derived from the children inputs by applying either rules SP1 and SP2. When updates occur to the base tuples, changes are propagated up the tree to the root. For example, when the cost of $\#link(\mathbf{a},b,5)$ is updated from 5 to 1, $path(\mathbf{a},b,e,[a,b,e],2)$ and $path(\mathbf{e},a,[e,a,b,e],3)$ are re-derived and replace the previous tuples. Similarly, the deletion of $link(\mathbf{b},e,1)$ leads to the deletion of $path(\mathbf{b},e,e,[b,e],1)$, $path(\mathbf{a},b,e,[a,b,e],2)$, and then $path(\mathbf{e},a,e,[e,a,b,e],3)$.

Let FP_p be the set of tuples derived using PSN under the bursty model, and FFP_p be the set of tuples that would be computed by PSN if starting from the quiesced state. In Appendix B, we prove the following theorem:

Theorem 3: *$FP_p = FFP_p$ in a centralized setting.*

The proof requires that all changes (inserts, deletes, updates) are applied in the same order in which they arrive. This is guaranteed by the FIFO queue of PSN and the use of timestamps.

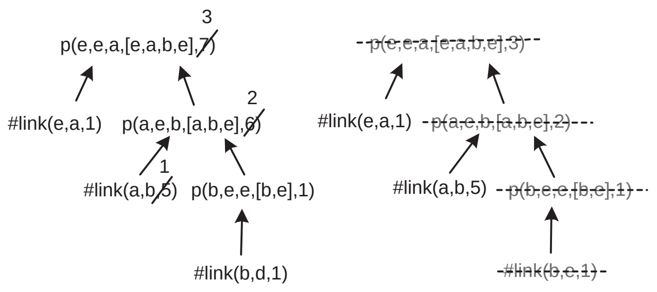

Figure 6: ***Derivation tree for derived path tuple from a to e. The left diagram shows updating the tree due to a change in base tuple*** $\#link(\mathbf{a},b,5)$***, and the right diagram shows the deletion of*** $\#link(\mathbf{b},e,1)$**.**

4.2 Distributed Semantics

In order for incremental evaluation to work in a distributed environment, it is essential that along any link in the network, there is a FIFO ordering of messages. That is, along any link literal `#link` (s,d), facts derived at node s should arrive at node d in the same order in which they are derived (and vice versa). This guarantees that updates can be applied in order. Using the same definition of FP_p and FFP_p as before, assuming the link FIFO ordering, in Appendix B, we prove the following theorem:

Theorem 4: *$FP_p = FFP_p$ in a distributed setting with FIFO links.*

The drawback of enforcing network link FIFO is that it increases the complexity and lowers the performance of the underlying network. The alternative adopted by network protocols is to maintain all tuples as *soft state*. In the soft state storage model, all data (base and derived tuples) has an explicit "time to live" (TTL), and facts (in our case base tuples) must be explicitly reinserted with their latest values and a new TTL or they are deleted. Reinsertion of base tuples leads to recomputation of query results, which in a quiescent network, leads to eventual consistency through the reinsertion of the facts from the quiescent state. The drawbacks of soft state are well known: recomputation can be expensive, and if done only periodically, the time to react to failures is a half-period on average. However, soft state is often favored in networking implementations because in a very simple manner it provides eventually correct semantics in the face of reordered messages, node disconnection, and other unpredictable occurrences.

5. QUERY OPTIMIZATIONS

We proceed to discuss a set of query optimization opportunities that arise in the declarative networking context. These include applications of traditional Datalog optimizations, as well as new techniques for multi-query optimization, result caching, and cost-based optimizations based on graph statistics. Some of these techniques—in particular the use of traditional Datalog optimizations and caching—were proposed in our previous work [24]. We present extensions to our basic techniques, as well as new avenues for optimization.

Compared to the relatively solid foundation of the previous discussion, our approach here is more speculative: we open up a number of broad issues, and in Section 6 we provide a taste of the potential benefits of most of them via a full-fledged implementation running on a sizable network testbed. However our intention here is not to "close the book" on any of these issues; much as in traditional database query optimization and execution, we expect that our techniques here for declarative networking will lead to significant work in a series of more focused investigations.

5.1 Traditional Datalog Optimizations

We first explore the applicability of three traditional Datalog optimization techniques: *aggregate selections*, *magic sets* and *predicate reordering*.

5.1.1 Aggregate Selections

A naïve execution of the *shortest-path* query computes all possible paths, even those paths that do not contribute to the eventual shortest paths. This inefficiency can be avoided with a query optimization technique known as *aggregate selections* [12, 36].

Aggregate selections are useful when the running state of a monotonic *AGG* function can be used to prune query evaluation. For example, by applying aggregate selections to the *shortest-path* query, each node only needs to propagate the most current shortest paths for each destination to neighbors. This propagation can be done whenever a shorter path is derived.

A potential problem with this approach is that the propagation of new shortest paths may be unnecessarily aggressive, resulting in wasted communication. As an enhancement, in the *periodic aggregate selections* scheme, a node buffers up new paths received from neighbors, recomputes any new shortest paths incrementally, and then propagates the new shortest paths periodically. The periodic technique has the potential for reducing network bandwidth consumption, at the expense of increasing convergence time. It is useful for queries whose input tuples tend to arrive over the network out of order in terms of the monotonic aggregate – *e.g.*, computing "shortest" paths for metrics that are not correlated with the network delays that dictate the arrival of the tuples during execution.

In addition, aggregate selections are necessary for the termination of some queries, as alluded to previously in Section 2. For example, with aggregate selections, even if paths with cycles are permitted, the *shortest-path* query will terminate, avoiding cyclic paths of increasing lengths.

5.1.2 Magic Sets and Predicate Reordering

The *shortest-path* query in our example computes *all-pairs* shortest paths. This leads to unnecessary overhead when only a subset of paths limited by sources and/or destinations is queried. This problem can be alleviated by applying two optimization techniques: *magic-sets rewriting* and *predicate reordering*.

Magic-Sets Rewriting: To limit query computation to the relevant portion of the network, we use a query rewrite technique, called *magic sets rewriting* [5]. The Magic Sets method is closely related to methods such as Alexander [30] and QSQ [22]. Rather than review Magic Sets here, we illustrate its use in an example: by modifying SP1 from the shortest-path query, the following computes only paths limited to destinations in the *magicDst* table.

#include(SP2,SP3,SP4)

SP1-D: path(**@S**,@D,@D,P,C) :- *magicDst(***@D***)*,`#link`*(***@S***,@D,C)*,
P = *f_concatPath*(link(**@S**,@D,C), nil).
Query: shortestPath(**@S**,@D,P,C).

Rule SP1-D initializes 1-hop paths for destinations whose *magicDst*(@**D**) is present in the *magicDst* table. This ensures that rule SP2 only propagates paths to selected destinations based on the *magicDst* table. The shortest paths are then computed as before using rules SP3 and SP4.

Predicate Reordering: The use of magic sets in the previous query is not useful for pruning paths from sources. This is because paths are derived in a *"Bottom-Up" (BU)* fashion starting from destination nodes, where the derived paths are shipped "backwards" along neighbor links from destinations to sources. Interestingly, switching the search strategy can be done simply by *reordering* the *path* and `#link` predicates. This has the effect of turning SP2 from a *right-recursive* to a *left-recursive* rule. Together with the use of magic sets, the following *magic-shortest-path* query allows filtering on *both* sources and *destinations*:

SP1-SD: pathDst(**@D**,@S,@D,P,C) :- *magicSrc(***@S***)*, `#link` (**@S**,@D,C),
P = *f_concatPath*(link(**@S**,@D,C), nil).
SP2-SD: pathDst(**@D**,@S,@Z,P,C) :- *pathDst(***@Z***,@S,@*Z_1*,*P_1*,*C_1*)*,
`#link` (**@Z**,@D,C_2), C := C_1 + C_2,
P = *f_concatPath*(P_1,link(**@Z**,@D,C_2)).
SP3-SD: spCost(**@D**,@S,min<C>) :- *magicDst(***@D***)*,
pathDst(**@D**,@S,@Z,P,C).
SP4-SD: shortestPath(**@D**,@S,P,C) :- spCost(**@D**,@S,C),
pathDst(**@D**,@S,@Z,P,C).

The query computes 1-hop paths starting from each *magicSrc* using rule SP1-SD. Rule SP2-SD then recursively computes new paths by following all reachable links, and stores these paths as `pathDst(`**`dst`**`, src, prevHop, pathVector, cost)` tuples at each destination. Rules SP3-SD and SP4-SD then filter relevant paths based on *magicDst*, and compute the shortest paths, which can then be propagated along the shortest paths back to the source node. In fact, executing the query in this *"Top-Down" (TD)* fashion resembles a network protocol called *dynamic source routing* [20] which is proposed for ad-hoc wireless environments, where the high rate of change in the network makes such targeted path discovery more efficient compared to computing all-pairs shortest paths.

5.2 Multi-Query Optimizations

In a distributed setting, it is likely that many related queries will be concurrently executed independently by different nodes. A key requirement for scalability is the ability to share common query computations (*e.g.*, pairwise shortest paths) among a potentially large number of queries. We outline two basic strategies for multi-query sharing in this environment: *query-result caching* and *opportunistic message sharing*.

Query-Result Caching. Consider the *magic-shortest-path* query where node *a* computes $shortestPath(\mathbf{a}, d, [a,b,d], 6)$ to node *d*. This cached value can be reused by all queries for destination *d* that pass through *a*, *e.g.*, the path from *e* to *d*. Currently, our implementation generates the cache internally, building a cache of all the query results (in this case *shortestPath* tuples) as they are sent back on the reverse path to the source node. Since the subpaths of shortest paths are optimal, these can also be cached as an enhancement. As ongoing work, we are exploring techniques for declaratively specifying the cache, and evaluating caching policies.

Opportunistic Message Sharing. In the previous example, we consider how different nodes (src/dst) can share their work in running the *same* query logic with different constants. Sharing across *different* queries is a more difficult problem, since it is non-trivial to detect query containment in general [9]. However, we observe that in many cases, there can be correlation in the message patterns even for different queries. One example arises when different queries request "shortest" paths based on different metrics, such as latency, reliability and bandwidth; *path* tuples being propagated for these separate queries may be identical modulo the metric attribute being optimized.

A strategy that we have implemented is *opportunistic message sharing*, where multiple outgoing tuples that share common attribute values are essentially joined into one tuple if they are outbound to the same destination and share several common attributes; they can be re-partitioned at the receiving end. This achieves the effects of jointly rewriting the queries in a fashion, but on an opportunistic basis: derivations are done in this combined fashion only in cases that are spatiotemporally convenient during processing. In order to improve the odds of achieving this sharing, outbound tuples may be buffered for a time and combined in batch before being sent.

As an alternative to this opportunistic sharing at the network level, one can achieve explicit sharing at a logical level, *e.g.*, using correlated aggregate selections for pruning different paths based on a combination of metrics. For example, consider running two queries: one that computes shortest latency paths, and another that computes max-bandwidth paths. We can rewrite these as a single query by checking two aggregate selections, *i.e.*, only prune paths that satisfy *both* aggregate selections.

5.3 Cost-Based Rewrites

Currently, queries are executed using a left- (BU) or right-recursive (TD) query expression (Section 5.1.2). Our main goal during query execution is *network efficiency* (*i.e.*, reducing the burden on the underlying network), which, typically, also implies faster query convergence. It is not difficult to see that neither BU nor TD execution is universally superior under different network/query settings. Even in the simple case of a shortest-path discovery query $shortestPath(@S, @D, P, C)$ between two given nodes $(@S, @D)$, minimizing message overhead implies that our query processor should prefer a strategy that restricts execution to "sparser" regions of the network (*e.g.*, doing a TD exploration from a sparsely-connected source *@S*).

We argue that *cost-based* query optimization techniques are needed to guarantee effective query execution plans. While such techniques have long been studied in the context of relational database systems, optimizing distributed recursive queries for network efficiency raises several novel challenges that we are exploring in our ongoing work. In the remainder of this section, we briefly discuss some of our preliminary ideas in this area and their ties with work in network protocols.

The Neighborhood Function Statistic. As with traditional query optimization, cost-based techniques must rely on appropriate *statistics* for the underlying execution environment that can drive the optimizer's choices. One such key statistic for network efficiency is the *local neighborhood function* $N()$. Formally, $N(X,r)$ is the number of distinct network nodes within *r* hops of node *X*. The neighborhood function is a natural generalization of the size of the transitive closure (*i.e.*, reachability set) of a node, that can be estimated locally (*e.g.*, through other recursive queries running in the background/periodically). $N(X,r)$ can also be efficiently *approximated* through approximate-counting techniques using small (log-size) messages [31]. To see the relevance of $N()$ for our query-optimization problem, consider our example $shortestPath(@s, @d, P, C)$ query, and let $\mathtt{dist}(s,d)$ denote the distance of *s*, *d* in the network. A TD search would explore the network starting from node *s*, and (modulo network batching) result in a total of $N(s, \mathtt{dist}(s,d))$ messages (since it reaches all nodes within a radius of $\mathtt{dist}(s,d)$ from *s*). Note that each node only forwards the query message once, even though it may receive it along multiple paths. Similarly, the cost for a BU query execution is $N(d, \mathtt{dist}(s,d))$. However, neither of these strategies is necessarily optimal in terms of message cost. The optimal strategy is actually a *hybrid scheme* that "splits" the search radius $\mathrm{dist}(s,d)$ between *s* and *d* to minimize the overall messages; that is, it first finds r_s and r_d such that:

$$(r_s, r_d) = \arg \min_{r_s + r_d = \texttt{dist}(s,d)} \{ N(s, r_s) + N(d, r_d) \},$$

and then runs concurrent TD and BU searches from nodes s and d (with radii r_s and r_d, respectively). At the end of this process, both the TD and the BU search have intersected in at least one network node, which can easily assemble the shortest (s,d) path. The above search strategy can be easily implemented as a rewrite using simple *NDlog* rules. While the above optimization problem is trivially solvable in $O(\texttt{dist}(s,d))$ time, generalizing this hybrid-rewrite scheme to the case of multiple sources and destinations raises difficult algorithmic challenges. And, of course, adapting such cost-based optimization algorithms to work in the distributed, dynamic setting poses systems challenges. Finally, note that neighborhood-function information can also provide a valuable indicator for the utility of a node as a result cache (Section 5.2) during query processing.

Adaptive Network Routing Protocols. While we do not evaluate the above concepts in our experiments below, we note that the networking literature has considered adaptive routing protocols that strongly resemble our use of hybrid rewrites; hence, we believe this is an important area for future investigation and generalization. One interesting example is the class of *Zone-Routing Protocols* (ZRP) [16]. A ZRP algorithm works by each node precomputing *k-hop-radius* shortest paths to neighboring nodes (in its "zone") using a BU strategy. Then, a shortest-path route from a source to destination is computed in a TD fashion, using essentially the *magic-shortest-path* query described above, utilizing any precomputed shortest paths along the way. Each node sets its zone radius k adaptively based on the density and rate of change of links in its neighborhood; in fact, recent work [29] on adjusting the zone radius for ZRP-like routing uses exactly the neighborhood-function statistic.

6. EXPERIMENTS

We have prototyped our language, execution model, and some of our optimizations as modifications to the P2 system. Our prototype takes as input *NDlog* programs, performs rule localization, and generates a dataflow graph consisting of P2 *elements*. Each element is a node in the dataflow graph, and performs tasks such as queuing, network processing and traditional relational operations like joins and aggregations.

The generated execution plan is structurally similar to Figure 5, where there are rule strands comprising chains of elements. Each rule strand takes as input a queue, corresponding to new tuples for each strand. Our current implementation uses the PSN algorithm at the tuple granularity. A new tuple is dequeued and processed by the rule strand to generate new tuples which are then enqueued at the same node or sent as a network message for further processing at another node.

Beyond validating our language and implementation, the main goal of our evaluation is to verify the effectiveness of several of the proposed optimizations. In evaluating our system, the main metrics that we use are:

Convergence time: The time taken for the query execution to generate all the query results.

Communication overhead: The number of bytes transferred for each query. We consider both aggregate communication overhead (MB), as well as per-node bandwidth (kBps).

In summary, we find:

1. The aggregate selections optimization reduces communication overhead. Using *periodic aggregate selections* reduces this overhead further.
2. The use of magic sets and predicate reordering reduces communication overhead when only a limited number of paths are queried.
3. Multi-query sharing techniques such as query result caching and opportunistic result caching demonstrate the potential to reduce communication overhead when there are several concurrent queries.
4. On a network with bursty updates, incremental query evaluation techniques can recompute paths at a fraction of the cost of recomputing the queries from scratch.

6.1 Setup

Our experiments are conducted by running our modified P2 on 100 nodes on the Emulab [10] testbed. This testbed emulates realistic latency and bandwidth constraints seen on the Internet, yet provides repeatable experiments under a controlled environment. As input to the Emulab testbed, we use transit-stub topologies generated using GT-ITM [1], a package that is widely used to model Internet topologies. Our topology has four transit nodes, eight nodes per stub and three stubs per transit node. Latency between transit nodes is 50 ms, latency between transit nodes and their stub nodes is 10 ms, and latency between any two nodes in the same stub is 2 ms. The link capacity is set to 10 Mbps.

We construct an overlay network over the base GT-ITM topology where each node is assigned to one of the stub nodes. Each overlay node runs P2 on one Emulab machine, and picks four randomly selected neighbors. Each node has four link tuples, one for each neighbor. Each link tuple has metrics that include latency (based on the underlying GT-ITM topology), reliability (link loss correlated with latency), and a randomly generated value.

We base our workload primarily on routing protocols [24], and benchmark four variants of the same *shortest-path* query, differing in the link metric each seeks to minimize. On all our graphs, we label these queries by their link metric: *Hop-Count*, *Latency*, *Reliability* and *Random*, respectively. Note that *Random* serves as our stress case: we expect it to have the worst performance among all queries, because aggregate selections are less likely to be effective when the aggregate metric is uncorrelated with the network latency, which determines tuple arrival order during query execution.

6.2 Aggregate Selections

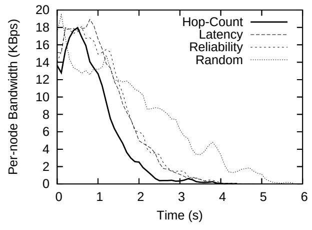

Figure 7: ***Per-node Bandwidth (kBps).***

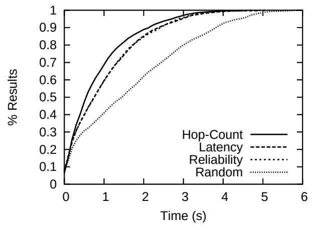

Figure 8: ***Query results over time (seconds).***

We first investigate the effectiveness of aggregate selections for different queries. Figure 7 shows the per-node bandwidth usage against time for the four queries. Figure 8 shows the percentage of eventual best paths completed against time. Our results show that *Hop-Count* converges the most quickly in 4.4 seconds, followed by *Latency* and *Reliability* in 4.9 seconds and 4.8 seconds respectively. *Random* has the worst convergence time of 5.8 seconds.

During query execution, the communication overhead incurred by all four queries shows a similar trend (Figure 7). Initially, the communication overhead increases as more and more paths (of increasing length) are derived. After it peaks at around 19*kBps* per-node, the communication overhead decreases, as fewer and fewer optimal paths are left to be derived. In terms of aggregate communication overhead, *Random* incurs the most overhead (4.1 MB), while *Hop-Count*, *Latency* and *Reliability* use 2.6 MB, 3.1 MB and 3.2 MB, respectively. The relatively poor performance of *Random* is due to the lack of correlation between the metric and network latency, leading to a greater tendency for out-of-order arrival of path tuples that results in less effective use of aggregate selections.

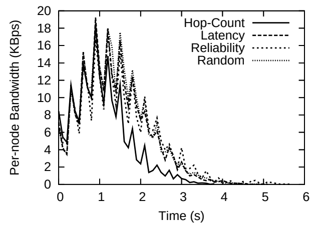

Figure 9: ***Per-node Bandwidth (kBps).***

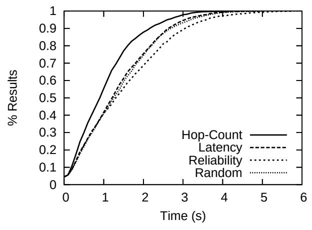

Figure 10: ***Query results over Time (seconds).***

The results in Figures 9 and 10 illustrate the effectiveness of the *periodic aggregate selections* approach, as described in Section 5.1.1. In particular, this approach reduces the bandwidth usage of *Hop-Count*, *Latency*, *Reliability* and *Random* by 17%, 12%, 16% and 29%, respectively. *Random* not only shows the greatest reduction in communication overhead, its convergence time also reduces from 5.8 seconds to 5 seconds.

6.3 Magic Sets and Predicate Reordering

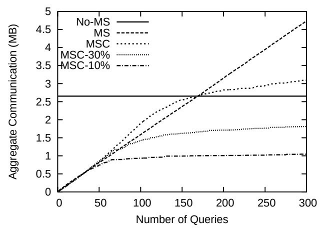

Figure 11: ***Aggregate communication overhead (MB) with and without magic sets and caching.***

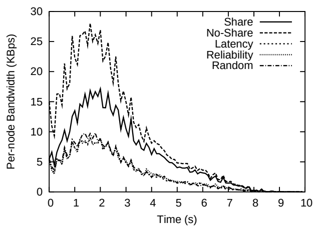

Figure 12: ***Per-node Bandwidth (kBps) for message sharing (300 ms delay).***

Next, we study the effectiveness of combining the use of magic sets and predicate reordering for lowering communication overhead when the queries are constrained by randomly chosen sources and destinations. Our workload consists of queries that request source-to-destination paths based on the *Hop-Count* metric. For each query, we execute the *magic-shortest-path* query (Section 5.1.2).

Figure 11 shows the aggregate communication overhead as the number of queries increases. The *No-MS* line represents our baseline, and shows the communication overhead in the absence of rewrites (this essentially reduces to computing all-pairs least-hop-count). The *MS* line shows the communication overhead when running the optimized query with no sharing across queries. When there are few queries, the communication overhead of *MS* is significantly lower than that of *NO-MS*. As the number of queries increases, the communication overhead of *MS* increases linearly, exceeding *No-MS* after 170 queries.

In addition, Figure 11 also illustrates the effectiveness of caching (Section 5.2). The *MSC* line shows the aggregate communication overhead for magic sets with caching. For fewer than 170 queries, there is some overhead associated with caching. This is due to false positive cache hits, where a cache result does not contribute to computing shortest paths. However, as the number of queries increases, the overall cache hit rate improves, resulting in a dramatic reduction of bandwidth. When limiting the choice of destination nodes to 30% (*MSC-30%*) and 10% (*MSC-10%*), the communication overhead levels of at 1.8 MB, and 1 MB, respectively. The smaller the set of requested destinations, the higher the cache hit rate, and the greater the opportunity for sharing across different queries. These results are consistent with the results obtained by Loo *et al.* [24] in a similar experiment, using the PIER [17] simulator.

6.4 Opportunistic Message Sharing

We study the impact of performing opportunistic message sharing across concurrent queries that have some correlation in the messages being sent. Figure 12 shows per-node bandwidth usage for running the queries on different metrics concurrently. To facilitate sharing, we delay each outbound tuple by 300ms in anticipation of possible sharing opportunities. The *Latency*, *Reliability* and *Random* lines show the bandwidth usage of each query individually. The *No-Share* line shows the total aggregate bandwidth of these three queries without sharing. The *Share* line shows the aggregate bandwidth usage with sharing. Our results clearly demonstrate the potential effectiveness of message sharing, which reduces the peak of the per-node communication overhead from 27 kBps to 16 kBps, and the total communication overhead by 34%.

6.5 Incremental Query Evaluation

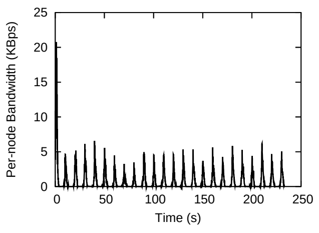

Figure 13: ***Per-node Bandwidth (kBps) for periodic link updates on latency metric (10s update interval).***

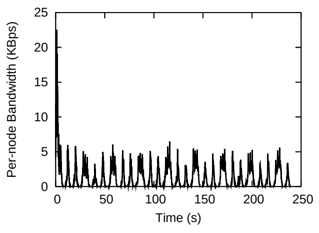

Figure 14: ***Per-node Bandwidth (kBps) for periodic link updates (interleaving 2s and 8s update interval).***

In our final experiment, we examine the overhead of incrementally maintaining query results in a dynamic network. We run the queries over a period of time, and subject the network to burst updates as described in Section 4. Each update burst involves randomly selecting 10% of all links, and then updating the cost metric by up to 10%. We use the shortest-path random metric since it is the most demanding in terms of bandwidth usage and convergence time.

Figure 13 plots the per-node communication overhead, when applying a batch of updates every 10 seconds. Two points are worth noting. First, the time it takes the query to converge after a burst of updates is well within the 5 second convergence time of running the query from scratch (Figure 10). This is reflected in the communication overhead, which increases sharply after a burst of updates is applied, but then disappears long before the next burst of updates (Figure 13). Second, each burst peaks at 6*kBps*, which is only 32% of the peak bandwidth and 26% of the aggregate bandwidth of the original computation. Our results clearly demonstrate the usefulness of performing incremental query evaluation in response to changes in the network, as opposed to recomputing the queries from scratch.

We repeat our experiment on a more demanding update workload (Figure 14), where we interleave update intervals that are 2 seconds and 8 seconds, the former interval being less than the from-scratch convergence time of 5 seconds. We observe that despite the fact that bursts are sometimes occurring faster than queries can run, bandwidth usage is similar to the less demanding update workload. When the update interval is 2 seconds, we notice periods of sustained bandwidth usage, however the peak usage remains at 6 kBps as before.

7. ADDITIONAL RELATED WORK

We mentioned most of the related work in the context of our discussion above. Here, we briefly mention some other related efforts.

We are not alone in our renewed enthusiasm for applications of recursive queries. There are other contemporary examples from outside the traditional database "market", including software analysis [37], trust management [6] and diagnosis of distributed systems [2]. Our concept of *link-restricted* rules is similar in spirit to *d3log* [19], a query language based on Datalog proposed for dynamic site discovery along web topologies.

Much research in the parallel execution of recursive queries [8] has focused on high throughput within a cluster. In contrast, our strategies and optimizations are geared towards bandwidth efficiency and fast convergence in a distributed setting. Instead of hash-based partitioning schemes that assume full connectivity among nodes, we are required to perform query execution only along physical network links and deal with network changes during query execution. There is also previous empirical work on the performance of parallel pipelined execution of recursive queries [32]. Our results extend that work by providing new, provably correct pipelining variants of semi-naïve evaluation.

In terms of distributed systems, the closest analog is the recent work by Abiteboul *et al.* [2]. They adapt the QSQ [22] technique to a distributed domain in order to diagnose distributed systems. An important limitation of their approach is that they do not consider partitioning of relations across sites as we do; they assume each relation is stored in its entirety in one network location. Further, they assume full connectivity and do not consider updates concurrent with query processing.

8. CONCLUSION

Our goal in this paper was twofold: to provide a solid database foundation for recent developments in declarative networking, and to open a number of database research directions in the area. We believe that our contributions here are significant on both fronts.

We started with the concept of *link-restricted rules*, which capture syntactically in *NDlog* the notion that query messages are constrained to travel along direct links between nodes in a network. This in turn led to successive refinements of semi-naïve evaluation that deal efficiently with the asynchrony and delays intrinsic to a wide-area networking environment. We introduced techniques to incorporate updates immediately during execution, capturing the reactive nature of typical network protocols while offering meaningful semantic guarantees. We also discussed a number of query optimization techniques, and their applicability to the networking domain. Finally, we presented evaluation results from a distributed deployment involving 100 machines on the Emulab [10] network testbed, running prototypes of our optimization techniques implemented as modifications to the P2 system.

Our ongoing research is proceeding in several directions. First, we are exploring a complete query optimization architecture, as well as specific techniques beyond those of Section 5: additions to the cost-based optimizations of Section 5.3 including the possibility of using random walks driven by statistics on graph expansion; adaptive query processing techniques to react to network dynamism; and multi-query optimizations motivated by more complex overlay networks. Second, we plan to incorporate negation into our model and implementation [18], which raises interesting challenges for pipelining and dynamic data. Third, a key selling point of declarative languages in the networking community is the promise of static program checks for desirable network protocol properties; we are considering techniques from the Datalog literature in this regard (*e.g.*, [21]) and expect that the particulars of link-restricted rules can be of use as well. Finally, we intend to aggressively pursue these ideas in the context of serious networking applications, *e.g.*, overlay networks like distributed hash tables, application-level multicast protocols, and virtual private networks.

We have been pleased in this work to see that the enthusiasm in the networking community for declarative languages can provide more than just a well-motivated application area for recursive queries; it appears to spark a host of new database research challenges in what was considered a very mature area. We are optimistic about the potential for additional significant results in this domain, in terms both of theoretical work and systems challenges.

9. REFERENCES

[1] GT-ITM. http://www.cc.gatech.edu/projects/gtitm/.

[2] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of Asynchronous Discrete Event Systems - Datalog to the Rescue! In *ACM PODS*, 2005.

[3] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Prog, 4(3):259–262*, 1987.

[4] F. Bancilhon. Naive Evaluation of Recursively Defined Relations. *On Knowledge Base Management Systems: Integrating AI and DB Technologies*, 1986.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *SIGMOD*, 1986.

[6] M. Y. Becker and P. Sewell. Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[7] P. A. Bernstein, U. Dayal, D. J. DeWitt, D. Gawlick, J. Gray, M. Jarke, B. G. Lindsay, P. C. Lockemann, D. Maier, E. J. Neuhold, A. Reuter, L. A. Rowe, H.-J. Schek, J. W. Schmidt, M. Schrefl, and M. Stonebraker. Future Directions in DBMS Research. *SIGMOD Record*, 18(1):17–26, 1989.

[8] F. Cacace, S. Ceri, and M. A. W. Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases*, 1(4):337–382, 1993.

[9] D. Calvanese, G. D. Giacomo, and M. Y. Vardi. Decidable Containment of Recursive Queries. In *ICDT*, 2003.

[10] Emulab. http://www.emulab.net.

[11] N. Feamster and H. Balakrishnan. Correctness properties for Internet routing. In *Allerton Conference on Communication, Control, and Computing*, Sept. 2005.

[12] F. Furfaro, S. Greco, S. Ganguly, and C. Zaniolo. Pushing Extrema Aggregates to Optimize Logic Queries. *Inf.Sys.*, 27(5):321–343, 2002.

[13] Overcoming barriers to disruptive innovation in networking. Report of NSF Workshop, Jan. 2005.

[14] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, 1993.

[16] Z. J. Haas. A New Routing Protocol for the Reconfigurable Wireless Networks. In *IEEE Int. Conf. on Universal Personal Communications*, 1997.

[17] R. Huebsch, B. Chun, J. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR*, 2005.

[18] Jeffery Ullman. Assigning an Appropriate Meaning to Database Logic with Negation. *Computers as Our Better Partners*, pages 216–225, 1994.

[19] T. Jim and D. Suciu. Dynamically Distributed Query Evaluation. In *PODS*, 2001.

[20] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, volume 353. 1996.

[21] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A Framework for Testing Safety and Effective Computability. *J. Comp. Sys. Sci. 52(1):100-124*, 1996.

[22] Laurent Vieille. Recursive Axioms in Deductive Database: The Query-Subquery Approach. In *1st International Conference on Expert Database Systems*, 1986.

[23] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[24] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[25] L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan-KaufMann, 2003.

[26] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. In *HotNets-III*, 2004.

[27] R. Ramakrishnan, K. A. Ross, D. Srivastava, and S. Sudarshan. Efficient Incremental Evaluation of Queries with Aggregation. In *SIGMOD*, 1992.

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

[29] V. Ramasubramanian, Z. J. Haas, and E. G. Sirer. SHARP: A Hybrid Adaptive Routing Protocol for Mobile Ad Hoc Networks. In *ACM MobiHoc*, 2003.

[30] J. Rohmer, R. Lescoeur, and J. M. Kerisit. Alexander Method - A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing 4:522-528*, 1986.

[31] C. R.Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. In *ACM SIGKDD*, pages 102–111, 2002.

[32] J. Shao, D. A. Bell, and M. E. C. Hull. An Experimental Performance Study of a pipelined recursive query processing strategy. In *International Symposium on Databases for Parallel and Distributed Systems*, 1990.

[33] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Distributed Monitoring and Forensics in Overlay Networks. In *Eurosys*, 2006.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[35] M. Stonebraker and J. M. Hellerstein, editors. *Readings in Database Systems, Third Edition*. Morgan Kaufmann, San Francisco, 1998.

[36] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *VLDB*, 1991.

[37] J. Whaley and M. S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, 2004.

APPENDIX

A. PROOFS FOR PIPELINED SEMI-NAÏVE

Symbol	Representation
t	A tuple generated at any iteration.
t^i	A tuple generated at the i^{th} iteration.
p_k	The table corresponding to the k^{th} recursive predicate in the rule body.
b_k	A table for the k^{th} base predicate in the rule body.
$FP_S(p)$	Result set for p using SN evaluation.
$FP_P(p)$	Result set for p using PSN evaluation.
$FP_S^i(p)$	Result set for p using SN evaluation at the i^{th} iteration or less.
$FP_P^i(p)$	Result set for p using PSN evaluation for all p tuples that are marked with iteration number i or less.

Table 1: Proof Notation

In our proofs, we use the notation in Table 1. Consider a rule with n recursive predicates p_1, p_2,..., p_n and m base predicates:
$p :- p_1, p_2, ..., p_n, b_1, b_2, ..., b_m$.

For the purposes of the proof of Theorem 1, we assume that there is a unique derivation for each tuple t.

Claim 2 *$\forall t^i \in FP_S^i(p), \exists t_j \in FP_S^{i-1}(p_j)$ s.t. $t :- t_1, t_2, ..., t_n, b_1, b_2, ..., b_m$ $\wedge\ t \notin FP_S^{i-1}(p)$. Same for FP_P.*

Theorem 1 *$FP_S(p) = FP_P(p)$*

Proof: (By induction). The base case $FP_S^0(p) = FP_P^0(p)$ is trivial since this is the initial set of input p_0 tuples. Assume inductively $FP_S^{i-1}(p) = FP_P^{i-1}(p)$ is true, we show that $FP_S^i(p) = FP_P^i(p)$ using the following two lemmas below. □

Lemma 1 *$FP_S^i(p) \subseteq FP_P^i(p)$*

Proof: Consider tuple $t^i \in FP_S^i(p)$ derived using SN evaluation $t :- t_1, t_2, ..., t_n, b_1, b_2, ..., b_m$. By Claim 2, $t_j \in FP_S^{i-1}(p_j) \wedge t \notin FP_S^{i-1}(p)$. One of the input t_j's (t_k) must be in $\triangle p_k^{old}$ in the SN algorithm. $t_k^{i-1} \in FP_S^{i-1} \Rightarrow t_k^{i-1} \in FP_P^{i-1}$. By the PSN algorithm, t_j^{i-1} must have been enqueued, hence generating t^i. So $t^i \in FP_S^i$. □

Lemma 2 *$FP_P^i(p) \subseteq FP_S^i(p)$*

Proof: Consider a tuple $t^i \in FP_S^i(p)$ derived using modified PSN evaluation $t :- t_1, t_2, ..., t_n, b_1, b_2, ..., b_m$. From claim 2, $t_k \in FP_P^{i-1}(p_k)$ $\wedge\ t \notin FP_P^{i-1}(p)$. By the PSN algorithm, one of t_j's (t_k) is $\triangle t_k^{old,i-1}$. This means that $t_k^{i-1} \in FP_S^{i-1}(p_k) \Rightarrow t_k^{i-1} \in \triangle p_k^{old}$ in the i^{th} iteration of the SN algorithm. This will result in the rule being used to generate t in the i^{th} iteration. Hence, $t^i \in FP_S^i$. □

If there are multiple derivations for the same tuple, we can apply the same proof above for Theorem 1 using the following modified PSN: if there are two derivations t^i and t^j ($j > i$) for the same tuple, the modified PSN algorithm guarantees that t^i is generated by enqueuing t^i even if t^j was previously generated. Note that the modified PSN algorithm leads to repeated inferences, but generates the same results as PSN.

Theorem 2 *There are no repeated inferences in computing $FP_P(p)$.*

Proof: For linear rules, the theorem is trivially true since we only add a new derived tuple into the PSN queue if it does not exist previously. This guarantees that each invocation of the rule is unique

For non-linear rules, we continue from Theorem 1's proof. Let $ts(t)$ be the sequence number or timestamp of derived tuple t. Following the proof for Lemma 1, only the k^{th} rule, where $ts(t_k^{i-1}) = max(ts(t_1^{i-1}), ts(t_2^{i-1}), ..., ts(t_n^{i-1}))$ will be used to generate t_0^i at the inductive step, ensuring no repeated inferences. □

B. PROOFS FOR BURSTY UPDATES

Let E be the set of all extensional tuples that appear during the execution of a program. Let D be the set of all tuples that can be derived from E (we assume $E \subseteq D$ for simplicity). A tuple $t \in D$ derived by the rule t:-$t_1, t_2, ..., t_n$ has a corresponding *tree fragment*, with parent t and children t_j. The *derivation tree* for D is built by assembling the tree fragments for all possible derivations of tuples in D. We distinguish the multiple tree fragments for multiple derivations of t, but to simplify notation, we use $t, t_1, \ldots$ to name tree nodes. Leaves of this tree are elements of E.

A series of insertions and deletions to the extensional relations is modeled as a sequence of values $t(0), \ldots, t(j)$ for each $t \in E$, where 1 means present and 0 means absent. Similarly, for all tree nodes t, we remember the sequence of values (presence or absence) assigned to t by the PSN algorithm after each child change. We write $t(\infty)$ to represent the value of t once the network has quiesced.

Let t be a tree node whose children are $t_1, t_2, ..., t_n$.

Claim 3 *Along any tree edge $t_k \rightarrow t$, value changes are applied in the order in which t_k's change. This property is guaranteed by PSN's FIFO queue.*

Lemma 3 *$t(\infty)$ is derived using $t_1(\infty), \ldots, t_n(\infty)$.*

Proof: (By induction) $t(0)$ is computed from the initial values of its children. Assume inductively that $t(j-1)$ is derived based on the $(j-1)^{th}$ change in its children. If child t_k changes, $t(j)$ is rederived, and based on Claim 3, reflects the latest value of t_k. Hence, $t(\infty)$ is derived from the last value of all its children. □

Let FP_p be the set of tuples derived using PSN under the bursty model, and FFP_p be the set of tuples that would be computed by PSN if starting from the quiesced state.

Theorem 3 *$FP_p = FFP_p$ in a centralized setting.*

Proof: We write $t(\omega)$ for the values derived by PSN when its starting state is $e(\infty)$ for $e \in E$. If $\forall t \in D$'s derivation tree, $t(\omega) = t(\infty)$ then $FP_p = FFP_p$. We prove this by induction on the height of tuples in the derivation tree. We define D_i to be all nodes of D's derivation tree at height i, with $D_0 = E$.

In the base case, $\forall t \in D_0$, $t(\infty) = t(\omega)$ by definition of the base tuple values. In the inductive step, we assume that $\forall j < i$, $\forall t \in D_j$, $t(\infty) = t(\omega)$. Consider $t \in D_i$. Based on Lemma 3, $t(\infty)$ will be derived from the $t_k(\infty)$ values of its children, which by induction are equal to $t_k(\omega)$. Hence $t(\infty) = t(\omega)$. □

Claim 4 *As long as all network links obey FIFO for transmitted messages, Claim 3 is true for any children of t that are generated using link-restricted Datalog rules.*

Theorem 4 *$FP_p = FFP_p$ in a distributed setting.*

Proof: With Claim 4, the proof is similar to that of Theorem 3. □