Declarative Networking with Distributed Recursive Query Processing

Anonymous (Paper #114)

ABSTRACT

There have been recent proposals in the networking and distributed systems literature on declarative networks, where networks are declaratively specified using a recursive query language. This represents a significant new application area for database recursive query processing technologies. In this paper, we extend upon these recent work in the following ways. First, we define and motivate the need for a class of topology-restricted recursive queries that has sufficient expressive power for a large class of queries necessary to enable declarative networks. Second, we describe how these queries can be executed in a distributed asynchronous fashion using SYSTEMX, a distributed relational query processor, and relate these execution strategies to the behavior of network protocols. Third, we describe eventual consistency semantics of recursive queries in distributed environments where query execution happens while the underlying data is changing. Last, we study a number of query optimization techniques adapted from existing recursive query optimizations and introduced by the uniqueness of this environment, and lay out a number of query optimization and semantics challenges for future work.

1. INTRODUCTION

Proposals have recently emerged in the networking and distributed systems literature to specify network routing algorithms using a declarative recursive query language. These have included expressing routing protocols for extensible IP routers [13] and rapid implementation of overlay networks [12]. Network behavior is specified as a set of queries, which are then optimized and executed using a distributed query engine.

These proposals are part of a larger recent trend in networking towards network routing architectures which are highly programmable, often on a per-application basis. One motivation for this development is to overcome barriers to the evolution of the Internet architecture by deploying overlay networks [15] as diverse operational prototypes of a future design; another is to allow modifications of existing routing protocols which admit proofs of desirable properties such as convergence and stability [7].

Declarative recursive query-based approaches are one of the most promising avenues for dealing with the complexity of this vision. The forwarding tables in network routing nodes are regarded as a view over changing ground state (network links, nodes, load, operator policies, etc.), and this view is maintained by the continuous evaluation of a distributed query over this state.

Early results are highly promising, for example Loo et. al. [12] show an executable implementation of the Chord [19] lookup overlay in 47 Datalog rules, versus several thousand lines of C++ for the original version. The approach promises the flexibility to support a wide range of routing protocols, with the ability to easily customize and compose protocols, and check safety properties. This represents a significant new application area for database query processing technologies, and provides a compelling application for the considerable body of research into recursive query processing.

So far, published work on declarative networking has concentrated on demonstrating the feasibility of the advantages of the approach, with less emphasis on the rigor that characterizes the database literature. In this paper, we lay out a number of research problems in this space from a database perspective: data and query semantics, query optimization, and execution problems, together with some initial solutions we are exploring in the context of the SYSTEMX distributed query processor.

The main contributions of paper are as follows:

- Propose and motivate topology restricted recursive queries for specifying declarative networks
- Describe distributed asynchronous semi-naive evaluation of recursive queries, and relate that to network protocols
- Data and query semantics in distributed environments.
- We study a number of query optimization techniques and lay out a number of query optimization challenges for future work.
- Prototype implementation using SYSTEMX, experimented on a realistic network testbed.

The organization of the paper is as follows. In Section 2, we start with a review of Datalog, introduce the query and data models of Distributed Datalog, and demonstrate the use of a Distributed Datalog query for implementing the well-known path vector protocol. In Section 3, we describe in details the steps required to generate the query execution plan. We then discuss query semantics in dynamic networks (Section 4). We then propose a number of query optimization techniques in Section 5. We then present experimental results (Section 6) and conclude.

2. DATA AND QUERY MODEL
We first provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman’s survey [17]. 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 “\( q_1, q_2, ..., q_n \) 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, as we discuss below). 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 for each rule to be specified with the Datalog program; in the absence of explicit information, the primary key is the full set of attributes in the relation. In our schema definitions, we underline the primary key attributes of a table (unless the primary key is the full set of attributes). 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 \( f_{\text{concatPath}} \) 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 in 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 \( \text{link}(\text{src}, \text{dst}, \text{cost}) \). R1 and R2 are used to derive “paths” in the graph, represented as tuples in the derived (“intensional”) relation \( \text{path}(\text{src}, \text{dst}, \text{nextHop}, \text{pathVector}, ...) \). The \text{src} and \text{dst} fields represent the endpoints of the path; the \text{pathVector} is a string encoding the full path. The \text{nextHop} field will only become useful to us in later discussion; we can ignore it here. Given the \text{path} relation, Rules R3 and R4 compute the shortest paths as the derived relation \( \text{shortestPath}(\text{src}, \text{dst}, \text{pathVector}, \text{cost}) \). R3 derives the relation \( \text{spCost}(\text{src}, \text{dst}, \text{mincost}) \) that computes the minimum cost for each \((\text{src,dst})\) group for all input paths. The rule \text{Query} specifies shortestPath tuples as the result tuples.

\begin{verbatim}
R1: path(S,D,D,P,C) :- link(S,D,C),
    P = f_{\text{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_{\text{concatPath}}(link(S,Z,C_1), P_2).
R3: spCost(S,D,minCost) :- path(S,D,P,C).
R4: shortestPath(S,D,C) :- spCost(S,D,C), path(S,D,Z,P,C).
Query: shortestPath(S,D,P,C).
\end{verbatim}

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 two “\( Z \)” fields in \( \text{link}(S,Z,C) \) and \( \text{path}(Z,D,Z_2,P_2,C_2) \) 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) under the assumption that all costs are \( > 0 \).

### 2.1 Network Datalog

In this section, we will 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 model and compute on physical network graphs. In describing our model, we will utilize the NDlog query shown in Figure 1 that is used for distributed computation of shortest paths.

![Figure 1: Shortest-Path Query](image)

One of the novelties of our setting, from a database perspective, is that data is distributed and relations may be partitioned across sites. In NDlog, we want to be able to express the physical location of tuples within the query language, in order to control data placement and movement explicitly. First, we introduce a data type \( \text{address} \), which is used to describe a network location. Names of address variables and constants are prepended with “\( @ \)”. Then, we introduce 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 will highlight it in \( \textbf{bold} \) for clarity. For example, the location specifier of \( \text{#link}(S,D,C) \) is \( @S \).

Both traditional parallel query processors and more recent distributed query engines like PIER [10] assume complete connectivity between nodes: 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 to achieve this, while PIER uses a routing overlay based on a distributed hash table to simulate the same conditions.

Our network setting is different, since the goal of networking is to provide full end-to-end connectivity rather than assuming it a priori. Consequently, we assume physical connectivity is incomplete (the normal case in a link-based network), and impose the constraint that messages (e.g., tuples participating in distributed joins) can only be exchanged by nodes that have physical connectivity.

Again, we want to expose this in the query language to facilitate the specification of messaging in the network graph. Hence we introduce the following notion:

**Definition 2** A link relation is a stored (“extensional”) relation \( \#link(\text{src}, \text{dst}, ...) \) 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...
Given that we will be executing queries across network links, it will be 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. Rule SP1, SP3 and SP4 are local rules. SP2 is a non-local rule since the *link* and *path* body predicates are stored at different locations.

In NDlog, the evaluation of any rule has to depend only on communication among the physical links. We introduce a simple syntactic constraint that achieves this:

**Definition 4** A link-restricted rule has the following properties:

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

This syntax precisely captures the requirement that we operate directly on network whose link connectivity is not a full mesh. The following is an example of a link-restricted rule:

\[
p(\text{@Y}, \ldots) \leftarrow \text{#link}(\text{@X}, \text{@Y}, \ldots), p(\text{@X}, \ldots), 
p(\text{@X}, \ldots), \ldots, p_n(\text{@X}, \ldots).
\]

This rule body of this example is executed at @X and the resulting \( p \) tuples are sent to @Y, preserving the communication constraints along links. Note that this example’s body predicates all have the same location specifier: @X, the source of the link. By contrast, rule SP2 of Figure 1 has some relations whose location specifier is the source, and others whose location specifier is the body. A query rewrite technique described in Section 3 transforms such rules to ensure all body predicates are at the same location; we will apply this rewrite to all eligible rules to simplify our execution.

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

**Definition 5** A 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. **Link relations:** The program includes at least one link relation. 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:** All non-local rules in the program are link-restricted by exactly one of the link relations.

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, each node initializes all their local path tables to 1-hop paths using SP1. In the 2nd iteration, using SP2, each node takes the input paths generated from the previous iteration, and computes 2-hop paths, which are then propagated to its neighbors. E.g., *path*(a, b, [a, b], 2) is generated at node b using *path*(b, d, d, [b, d], 1) from the 1st iteration, and propagated to node a.

As the paths are being computed, the shortest paths are also incrementally computed. For example, node a computes *path*(a, b, b, [a, b], 5) using rule SP1, and then sets its shortest path to *shortestPath*(a, b, [a, b], 5) using rule SP4. In the next iteration, node a receives *path*(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*(a, b, [a, b], 2) replaces the previous value.

In the distributed implementation, the use of an optimization (described in Section 5.1) requires each node to only propagate its current shortest path to its neighbor node. In that case, each node only needs to maintain the current shortest path to every destination reported by each neighbor. In this case, the primary key for *paths* is (src, dst, nextHop).

### 2.3 Expressiveness

It was argued by Loo et al. that executing a shortest path distributed Datalog query resembles the distributed computation of the well-known path vector [14] protocol itself. In their work [12, 13], Loo et al. describe Datalog-like programs for a variety of networking tasks, including standard routing protocols such as distance vector [14] and dynamic source routing [11], and more complex overlay network constructions such as multicast trees and overlay ring formation in Chord [19]. We note that NDlog is expressive enough for many of these programs, and provides the joint advantages of having clear semantics as described above (something that is not available in the Overlog language described in [12]) and a clearly defined link-restricted implementation as described below.
from a Datalog program. For ease of exposition, 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 [2,5] evaluation mechanism that ensures no redundant evaluations. As a quick review, in semi-naïve (SM) 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 as rule SP2-1:

**Rule SP2-1:** \( \Delta p^\text{new}_k(\exists \text{link}(@Z, @S,C_1)) \),

\( \Delta \text{path}^\text{old}(\exists \text{Z}, @D, Z_2, P_2, C_2) \), \( C = C_1 + C_2 \),

\( P = f_\text{concatPath}(\exists \text{link} (\exists \text{S}, @Z,C_2)) \).

Figure 3 shows the dataflow realization for rule SP2-1 using SYSTEMX. 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 \( \Delta \text{path}^\text{old} \) tuples generated in the previous iteration (\( \Delta \text{path}^\text{old} \)) to generate new paths (\( \Delta \text{path}^\text{new} \)) 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 implementation of multiple linear semi-naïve rule strands where each rule has the form \( \Delta p^\text{new} :: \Delta p^\text{old} \), \( b_1, b_2, ..., b_n \) in SYSTEMX, where \( b_1, ..., b_n \) are base predicates.

**Algorithm 1 Semi-naïve (SN) Evaluation in SYSTEMX**

```plaintext
while \( \exists B_k \text{size} > 0 \)
\( \Delta p^\text{old}_k = B_k \text{flush}() \)
execute strand for \( \Delta p^\text{new}_j \):

\( \exists D \in \Delta p^\text{new}_j \)

\( \text{if } t \not\in D \)
\( \text{then } p_j = p_j \cup t \)
\( B_j = B_j \cup t \)

end
```

In the algorithm, \( B_k \) denotes the buffer for \( p^\text{old}_k \). For each rule \( \Delta p^\text{new}_j :: \Delta p^\text{old}_k \), \( b_1, b_2, ..., b_n \), each iteration consists of flushing all existing \( p^\text{old}_k \) tuples from \( B_k \) generated in the previous iteration, executing the rule to generate new \( p^\text{new}_j \) tuples, which are inserted back into the buffer \( B_j \) after the rule has completed execution. 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 of the predicates are not partitioned in the same way around the network. A rule localization rewrite step ensures that all the predicates in the rule body are partitioned in the same way. This allows a rule body to be locally computable.

Consider rule SP2 from Section 2 where the link and path predicates are at different locations. These two predicates are joined by a common “@Z” address field. Figure 4 shows the corresponding logical query plan depicting the distributed join. The clouds represent the forwarding of tuples from one network node to another, and are labeled with the destination node. The first cloud (\#1link.Z) sends link tuples to the neighbor nodes indicated by their destination address fields, in order to join with matching path tuples stored by their source address fields. The second cloud (path.S) sends new path tuples computed from the join back based on the source address field for further processing.

Based on the above distributed join, rule SP2 can be rewritten into the two rules shown in Figure 5. In either rule body of SP2a and SP2b, the predicates have common location specifiers.

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

**Figure 5:** Localized Rewrite for rule SP2

**FIXME:** insert link restricted EXCHANGE discussion

The rewrite is achievable because the #1link and path predicates, although at different locations, share a common join address field. Based on Definition 4, we note that all link-restricted rules are localizable. In Algorithm 2, we summarize the general rewrite technique for an input set of rules R link-restricted rules. In the pseudocode, for simplicity, we assume that the locational specifiers of all the body predicates are sorted (@S followed by @D).

**Algorithm 2 Rule Localization Rewrite**

**proc** RuleLocalization(R)

```plaintext
while \( \exists \exists \text{rule } r \in R : h(@D,...) \) :
\( \exists \#1\text{link} (@S, @D,...) \),
\( p_1(@S,...)p_2(@S,...)p_3(@S,...)p_4(@S,...)p_5(@S,...) \),
R.remove(r)
R.add(h1(r)(@S,...) : \#1\text{link} (@S, @D,...) \),
R.add(h2(r)(@D,...) : h2(r)(@D,...)p_1(@D,...)p_2(@D,...) \)
R.add(h3(r)(@D,...) : h3(r)(@D,...)p_3(@D,...)p_4(@D,...) \)
end
```

After rule localization, we perform the semi-naïve rewrite, and then generate the rule strands shown in Figure 6. 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 #1link.rl tuples. Rule strand SP2b-2@Z takes the new #1link.rl tuples that received via the network, performs a join operation with the local path table to generate new paths.
3.3 Pipelined Evaluation

We further note that in our distributed implementation, the execution of rule strands can depend on tuples arriving via the network, and can also result in new tuples sent over the network. Recall that the traditional semi-naïve evaluation enforces one iteration to be completed before the next begins. In a distributed execution environment where messages can be delayed or lost, determining the completion of an iteration requires distributed consensus algorithms which is not practical.

In the same spirit as pipelined query processing [21, 22], the approach that we have taken is to make use of pipelined semi-naïve (PSN) evaluation, where new tuples that are generated from the semi-naïve rules are used immediately to compute tuples in the next iteration without waiting for the current iteration to complete. This ensures that no node is blocked, while waiting for computation to complete on another node.

Algorithm 3  Pipelined Semi-naïve (PSN) Evaluation

```
while \( |Q_k\text{. size} > 0 | \) do
  \( t_k^{old,i} \leftarrow Q_k\text{. dequeue}() \)
  execute strand for \( \Delta p^{new,i+1} \leftarrow t_k^{old,i}, \ldots \)
  foreach \( r^{new,i+1} \in \Delta f^{new,i+1} \) do
    if \( r^{new,i+1} \notin \Delta f \) then
      \( p_j = p_j \cup r^{new,i+1} \)
      enqueue(\( t_k^{new,i+1} \))
  end
end
```

Figure 3 shows the pseudocode for PSN for a general semi-naïve rule \( \Delta p^{new} \leftarrow \Delta p^{old}, \ldots \). Each tuple is denoted \( r \) has a superscript \( (old/new,i) \) where \( i \) is its corresponding iteration number in SN evaluation. In the pseudocode, there is a queue which acts as a buffer for new \( p_k \) tuples. A tuple is dequeued from \( Q_k \) and then used as input into the respective rule strand, generating new \( p_j \) tuples which are then pipelined 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 begin processing the tuple, hence the pipelined behavior of query processing.

Note that the pseudocode above enforces tuple-at-a-time evaluation, which is computationally inefficient. As an optimization, we can flush the queue as before to perform set operations, and pipeline any generated tuples to their respective queue. However, care has to be taken to ensure that the iteration number is maintained correctly, since the dequeued set may include tuples with different iteration numbers.

In order to compute rules with aggregation (such as SP3), we utilize incremental fixpoint evaluation techniques [16] that are amenable to pipelined query processing. When used for computing monotonic aggregates [18] such as \( \min, \max \) and \( \text{count} \), these aggregates can be incrementally recomputed by computing the new aggregate based on the current aggregate and each new input tuple. In Appendix A.2, we prove the correctness of PSN for linear recursive rules. We show that PSN is correct if it generates the same results as SN. We also PSN has to avoid duplicate evaluations like SN.

Pipelining non-linear rules with multiple recursive predicates have the extra complication of introducing duplicates. Consider the following rule with \( n \) recursive predicates and \( m \) base predicates

\[ \text{NL1}: \quad p : -p_1, p_2, \ldots, p_n, b_1, b_2, \ldots, b_m \]

In PSN evaluation, there are \( n \) semi-naïve rules generated, one for each recursive predicate. For the \( k^{th} \) recursive predicate, we define the \( k^{th} \) semi-naïve rule as follows:

\[ \text{NL1-}k: \quad \Delta p^{new} \leftarrow p_1, \ldots, \Delta p_k^{old}, \ldots, p_n, b_1, b_2, \ldots, b_m \]

Duplicates can arise when \( t_k \in p_j \) and \( t_k \in p_k \) are derived simultaneously. Based on the PSN algorithm, they are stored in their respective queues \( Q_j \) and \( Q_k \). When tuple \( t_k \) is dequeued from \( Q_j \), the \( j^{th} \) rule is executed using the pair \( t_j \) and \( t_k \) as inputs. Similarly, when tuple \( t_k \) is dequeued, the pair is used as inputs to the \( j^{th} \) rule. This results in duplicate evaluations.

To avoid such duplicate evaluations, we assign a monotonically increasing sequence number (or timestamp) to each tuple upon dequeuing for processing. Note that the use of sequence numbers on a per-tuple basis does not limit PSN to tuple-at-a-time evaluation. If we are dequeuing a set of tuples for processing, each tuple is assigned an individual sequence number. Each tuple used as input to a rule is processed using only tuples with older sequence number. This ensures no duplicate evaluations. We provide the proof of correctness and performance in Appendix A.3.

4. SEMANTICS IN A DYNAMIC NETWORK

In practice, the state of the network is constantly changing during query execution. While queries are executing to fixpoint, changes in the network may occur. Like network protocols, long-running queries should constantly perform recomputations to reflect the latest state of the 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. Here, the network never reaches a steady state for query results to reflect the state of the network.

- **Bursty Update Model.** In this idealized (but 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 (experiences no updates) for a time long enough to allow all the queries in the system to reach a fixpoint.

We focus on studying the semantics of our queries under the bursty update model. This model allows us to reason about eventual consistency of query results when the network quiesces.

We first begin by considering techniques for incremental recomputation of queries. These techniques are similar to those in incremental recursive view maintenance [8].

- **Insertion:** The insertion of a new tuple at any stage of processing can be naturally handled by semi-naive evaluation.

- **Deletion:** The deletion of a base tuple should lead to the invalidation of any tuples that were derived from that base tuple. Deletions are carried out incrementally via semi-naive evaluation by incrementally deriving all tuples that are to be deleted.

- **Update:** An update is treated as a deletion followed by an insertion. We allow updates where a new tuple replaces an existing tuple with the same primary key. An update may itself result in derivation of more updates that are propagated via semi-naive evaluation as well.

Some derived tuples may have multiple derivations. An example would be a query that simply computes reachability between pairs of nodes for a network where there are multiple possible paths between two nodes. For such tuples, we use the count algorithm [8] for keeping track of the number of derivations for each tuple, and only deleting a tuple when the count reaches 0.

Updates to aggregates are non-monotonic. The naïve solution is to recompute the entire aggregate from scratch whenever a new input tuple for the aggregate arrives. We utilize techniques used for incremental computation of aggregates [16] in the presence of updates. The arrival of new tuples may invalidate existing aggregates, and this technique bounds the re-evaluation cost to \(O(\log n)\) in time and \(O(n)\) in space for the min and max aggregates.

### 4.1 Centralized Example

![Derivation tree for the shortest-path query](image)

**Figure 7:** 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\((a,b,1)\), and the right diagram shows the deletion of \#link\((b,d,1)\).

Figure 7 shows a derivation tree for \(\text{path}(e,a,e,a,b,e,7)\) based on the shortest-path query. The leaves in the tree are the input \#link tuples. The root and the intermediate nodes are derived tuples recursively derived from the children by applying either rules SP1 and SP2.

In Appendix B, we prove that given any derivation tree, the eventual result at the root of the tree is based on the final state of the leaves. The proof is based on induction and requires the assumption that all changes (updates, deletions) to any previous input tuples are applied in the order in which they are generated. This property is guaranteed by the FIFO queue in PSN.

As an illustrative example using Figure 7, when updates occur to the base tuples, changes are propagated up the tree to the root. For example, when the link cost is updated from 5 to 1, \(\text{path}(a,b,e,[a,b,e],2)\) and \(\text{path}(e,a,[e,a,b,e],3)\) are rederive and replaces the previous tuples. Similar, the deletion of \#link\((a,b,1)\) leads to the deletion of \(\text{path}(b,e,e,[b,e],1)\) which in turns leads to the deletion of \(\text{path}(a,b,e,[a,b,e],6)\) and then \(\text{path}(e,a,e,[e,a,b,e],7)\).

### 4.2 Distributed Semantics

In order for incremental evaluation to work in a distributed environment, we enforce the following:

- **Network link FIFO.** Along any \#link \((s,d)\), messages derived at node s should arrive at node d in the same order in which they are derived. This guarantees that updates can be applied in order.

- **Primary Key.** We constraint that the primary key of each relation derived using localized rules that require sending messages to include the location specifier of its rule body predicates. This guarantees that multiple derived tuples propagated along different different paths are not considered updates of one another upon arriving at a node. We note that this property can be achieved by the shortest-path query, where the path tuples include the nextHop field as its primary key.

The drawback of enforcing network link FIFO is that this increases the complexity and performance of the underlying network, that has to ensure FIFO in the presence of link failures and congestion.

The alternative adopted by network protocols is to maintain all computed routing tables as soft-state. In our context, this means that all tuples have an associated timeout, and queries need to be periodically refreshed whenever base tuples are refreshed. This approach guarantees convergence in the bursty model. The drawback of this approach is that recomputation can be expensive and if done only periodically, affect the time to react to failures to a half-period on average.

### 5. QUERY OPTIMIZATIONS

In this section, we explore the use of a number of well-known query rewrites optimizations in the recursive query literature, and study the adaptations of these optimizations to distributed environments, and relate them to well-known network protocol optimizations. A subset of these optimizations have also been proposed in [13]. We also explore the use of multi-query optimizations that are well-motivated in our environment, an area not well-studied in either the recursive query processing or networking literature.

#### 5.1 Aggregate Selections

A naïve execution of the shortest-path query would derive all possible paths, even those that do not contribute to the shortest paths. This inefficiency can be avoided with a query optimization technique known as aggregate selections [6, 20].

In the context of the shortest-path query, applying aggregate selections means that each node only needs to propagate its current
shortest path to its neighbors whenever that path has changed.

In general, aggregate selections are useful when used for monotonic aggregates such as min and max. Aggregate selections are especially effectively when used to compute paths based on metrics that are correlated with network delay, hence ensuring that the shorter (“better”) path is always computed first. To take into account network delays, we can utilize periodic evaluation, where the shortest paths is only periodically propagated. This allows for more effective pruning.

In addition, aggregate selections are necessary for the termination of some queries. For example, without aggregate selections, if paths with cycles are permitted, the shortest-path query will run forever, generating cyclic paths of increasing lengths. Another example of a query that would benefit from aggregate selections is in computing the most reliable paths, based on multiplying the transmissions success rates of intermediate links and taking the max aggregate.

5.2 Magic Sets and Predicate Reordering

Reduce communication overhead where routes are requested for some sources and destinations. Too expensive to compute all-pairs shortest paths, especially when network is changing.

Src/Dst queries initiated at source or destination. Use left-linear recursion with magic sets initiated at either source or destination. Provide extra filter predicate at destination or source respectively. Since we limit communication along links, send back results on computed reverse path. If there are multiple destinations for each source, the link traversal can be shared for multiple destinations. Doing this predicate reordering to a right-linear recursive strategy gives us a protocol that resembles DSR. Aggregate selections apply at the destination.

INSERT DATALOG magic sets rewrite for shortest paths as a rewrite. Rewritten is done prior to rule localization and semi-naive rewrite.

5.3 Multi-query sharing

5.3.1 Results Caching

Magic sets allow us to limit evaluation by sources and destinations. We can improve upon this using caching of results along intermediate nodes.

INSERT DATALOG shortest-paths query that utilizes cache. Show that is a hybrid rewrite also.

5.3.2 Opportunistic Message Sharing

Opportunistically share messages between multiple concurrent overlapping path computations on different metrics/policies. Share path messages being propagated between neighbor nodes.

Scenarios where such sharing is possible:

- Queries with correlated link metrics. Example correlations: queries based on latency, loss-rates, bandwidth where these metrics are correlated.

- Queries on same metrics but different set of participating nodes (different src/dsts). If the participants overlap, there are sharing opportunities.

- Queries with common policies on what paths are “legal”. E.g. agreements to carry traffic for certain nodes may be common across different queries. Refer to policy example above.

Key mechanism: combine messages (for each src/dst path to be sent, if two messages from different queries differ only on a cost metric.

Some comparisons with logical sharing using query rewrites for sharing, combining aggregate selections. Sharing at logical level means the sharing is more limited. For example, common policies hard to detect.

5.4 Hybrid Rewrites

Combined top-down (magic sets) and bottom-up evaluation.

Combined the best of both worlds, can be expressed as hybrid rewrites. One example of this is the zone routing protocol used in wireless communication. Do a limited right-linear computation of all pairs, with filter at source (K-horizon all-pairs all paths). For each source/destination initiated query, check to see if it is within zone. If not, forward the query to all nodes at the peripheral of zone, and they recursively check within their zones. Best of both worlds if the horizon size is chosen right. Frame this as a “partial magic rewrite”.

INSERT in Datalog rules for doing a k-horizon shortest paths computation, and a source-routing mechanism that traverses zones (recursively route to peripheral nodes). Show that some rules are right recursive, others are left.

A right linear magic sets rule to do K-horizon all-pairs computation. Routing is done by the same left-linear approach with caching (as shown above), or recursively directs to perimeter of zone, then recursive routes until find the node responsible.

View of perimeter nodes. One rule. If in horizon, generate results, send back on reverse path. If not in horizon, forward query to perimeter nodes via links. Perimeter nodes receive query, check never received query before. If so, repeat the check.

As shown in [9], the size of radius depends on parameters like rate of change of paths, and the density of each node (number of neighbors in k hops). These can be captured for cost-based analysis using systems statistics.

6. EXPERIMENTS

To evaluate our solution, we have implemented a prototype system using SYSTEMX, a distributed relational query processor written in C++. The goal of our evaluation is two-fold. First, in Section 6.1, we validate query processing characteristics of running Distributed Datalog queries for both static and dynamic networks. Second, in Sections 6.2-6.3, we demonstrate the effectiveness of the query optimizations that we proposed in Section 5.

We evaluate our experiments on 100 nodes\(^1\) on the Emulab [4] testbed. This testbed allows us to emulate realistic latency and bandwidth constraints seen on the Internet, yet provide repeatability under a controlled environment. As input to the Emulab testbed, we make use of transit-stub topologies generated using GT-ITM [1] that is widely used to model Internet topologies. Our topology consists of four transit nodes, eight nodes per stub and three stubs per transit nodes. The latency between transit nodes is set to 50 ms, the latency between a transit and a stub node is 10 ms, and the 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 SYSTEMX resides in one Emulab machine, and maintains four randomly selected neighbors. Each node is then initialized with a link table, one for each neighbor. Each link tuple has additional metrics that include latency (computed based on the underlying GT-ITM topology), reliability (synthetically generated and correlated with latency), and a randomly generated value.

\(^1\)Our setup of 100 physical nodes is limited by the number of nodes on the Emulab testbed.
6.1 Query Processing

Our baseline query workload consists of running variants of the all-pairs best-path query to compute between all pairs of nodes, the paths with the least hop count, lowest latency and highest reliability. We also include a query for computing the shortest paths based on the random metric. By default, all our queries utilize PSN evaluation and also utilize the aggregate selections optimization.

Figure 8 shows the bandwidth utilization against time for running the four queries. Figure 9 shows the percentage of eventual best paths completed against time, and reflects the time taken for the query to converge (i.e., computed all the eventual best paths). Our results show that hop count converges most quickly in 4.4 sec, followed by latency and reliability in 4.9 sec and 4.8 sec respectively. The random metric has the worst convergence times in 5.8 sec.

In terms of bandwidth utilization, all four queries show an increase in bandwidth utilization as more paths (of increasing path length) are computed, peaking at 19Kbps (per-node utilization), and decreasing as more best paths are discovered and aggregate selections avoid sending of unnecessary paths. The random metric also utilizes the most bandwidth (4.1 MB), compared to only 2.3 MB and 3.1 MB for hop count and latency respectively. This validates our previous observation that aggregate selections are more effective for queries computing properties over metrics that are correlated with network latency.

6.1.1 Periodic Aggregate Selections

As an alternative, we explore the use of periodic pipelined query processing as described in Section 5. We execute the same four queries, but utilize periodic aggregate selections. The period between processing is set to 300 ms, i.e., a new “optimal” aggregate path (if available) is sent every 300 ms. Figure 10 and 11 shows that this approach leads to reduction in bandwidth. The bandwidth utilization reduces by 17%, 12% and 29% respectively for hop count, latency and random. The random metric shows the greatest reduction in bandwidth. On top of that, the convergence time for random reduces from 5.8 sec to 5 sec. This reduction is due to the effectiveness of periodic aggregate selections in avoiding sending unnecessary data, and queries over metrics that are not correlated with network latency shows the greatest reduction.

6.1.2 Incremental Query Evaluation on Bursty Networks

Next, we examine the overhead of performing incremental query evaluation on a dynamic network as described in Section 4. Figures 12 and 13 shows the bandwidth utilization for two different update workloads, where we emulate the bursty update model. Each burst of update involves randomly selecting 10% of all links, and then modifying the cost metric by up to 10% of their previous value.

Figure 12 utilizes has a batch of updates applied every 10 seconds for the best-path query on the random metric. We picked the random metric since it is the most demanding in terms of bandwidth utilization and convergence time. Notice that the update interval exceeds the convergence time of 5 sec. We make the following three observations. First, each update burst coincides with a sharp increase in bandwidth utilization, but the burst interval is bounded by the time required to compute the query from scratch. Second, each burst peaks at 6Kbps, which is only 32% of the peak bandwidth utilized. Third, on average, each burst utilizes 26% of the bandwidth required to compute the query from scratch. Our results clearly demonstrate the usefulness of performing incremental query evaluation in response to changes in the network, as opposed to re-computing the queries from scratch.

We repeat our experiment on a more demanding update workload, where we interleave update intervals that are 2 and 8 sec, the former being less than the convergence time of 5 sec. We observe that bandwidth utilization is similar to the less demanding update workload. When the update interval is 2 sec, we notice periods of sustained bandwidth utilization, however, peak utilization remains at 6Kbps as before. When the interval is increased to 8 sec, the query computation quiest as shown by the reduction of bandwidth utilization to 0 Kbps.

6.2 Magic Sets and Predicate Reordering

Next, we study the effects of using query optimization techniques to lower communication overhead when only a subset of paths are computed. Instead of computing all pairs, our workload consists of a collection of queries that request for source-to-destination paths based on the hop count metric. On each query, we perform the magic sets rewrite and predicate reordering as described in Section 5.2. Each query computes the shortest path between a pair of nodes, and the result tuple is sent back on the reverse path to the source.

Figure 14 shows the total bandwidth usage as the number of source/destination queries increases. The No MS line represents our baseline, and
shows the communication overhead for computing all pairs shortest paths without the use of any rewrites. MS shows the communication overhead for running the optimized query with no sharing across queries. When there are few queries, the communication overhead of MS is significantly lower than of NO MS, as the later computes many paths which were never requested. However, as the number of queries increases, the communication overhead increases linearly, exceeding All Pairs after 170 queries.

6.2.1 Magic Sets with Caching

Next, we study the impact of using magic sets in conjunction with caching, where computed shortest paths along intermediate nodes are opportunistically reused. In Figure 14, MS-100% shows the total bandwidth utilization for magic sets with caching. Interestingly, up to 170 queries, the cost of caching is more expensive than not caching. This is due to false positive cache hits, where some of the cache utilized on intermediate nodes do not contribute to the eventual shortest paths. However, as the number of queries increases, caching can reduce the bandwidth utilization dramatically.

By limiting the choice of destination nodes to 30% (10%) of nodes, the communication overhead levels off at 1.8 MB (1 MB). This is because, the smaller set of requested destinations, the higher the cache hit rate and the greater the opportunity for work-sharing.

6.3 Opportunistic Message Sharing

In our last experiment, we study the impact of performing opportunistic message sharing across concurrent queries that have some correlation between the messages being sent. We issue queries for three metrics (reliability, described in Section 6.1 simultaneously and measured the per-node bandwidth utilization over time.

Figure 15 shows our results. We repeat the experiment with a lower periodic interval of 50 ms, and observed that there is less sharing across queries. E.g., the peak per-node bandwidth utilization attained with sharing is 22 Kbps, which is only marginally lower compared to no sharing. This clearly demonstrates the usefulness in delaying query computations in time in order to facilitate sharing. Determining the correct wait interval at runtime is an interesting area for further exploration.

7. RELATED WORK

Query execution and optimization of recursive queries is a rich area of research. The survey paper [17] provides an excellent overview as well as references to query processing and optimization techniques. There has also been previous work on parallel execution strategies of recursive queries [3] over static data within a parallel cluster.

Find all the distributed/parallel Datalog literature and put it here. Recent application to other domains (petri-nets, language pointer analysis). Fill in related networking research work (keep this minimal).

8. CONCLUSION

Initial work. Lots of area of interesting work.

Future work: safety analysis (reason about protocols), dealing with negation semantics (global synchronization vs asynchronous corrections leading to eventual convergence), more optimizations (approximations, gossip), adaptive query optimizations.

APPENDIX

A. PIPELINED SEMI-NAÏVE PROOFS

A.1 Notation

In our proofs, $t$ refers to a tuple and $p$ refers to table $p$. The superscript letter $i$ refers to the height of the derivation tree, or the iteration number in SN evaluation, e.g., $t^i$ is generated at iteration $i$. The subscript letter $j$ and $k$ refer to indices of predicates, e.g., $b_j$ refers to the $j$th base predicate in the body of the rule LR1a. The letter $n$ and $m$ denote the number of recursive and base predicates respectively in a rule. $FP_{SN}$ denotes all tuples that are marked with derivation tree of $i$ or less. Similarly, $FP_{PSN}$ denotes all tuples that are marked with iteration number $i$ or less, based on the PSN algorithm (Figure 3).

A.2 Linear Recursive Proofs

Consider the following semi-naïve rewritten rule from Section 3: LR1a: $\Delta p^{new} \leftarrow \bigcup_{i} p_{old}^{ed}, b_1, b_2, \ldots, b_m$, where $p_1$ is a recursive predicate.

Theorem 1 $FP_{PSN}(p) = FP_{SN}(p) \land FP_{PSN}(p_1) = FP_{SN}(p_1)$

Proof: Refer to the proof of the more general Theorem 3

Theorem 2 There are no duplicate evaluations in computing $FP_{PSN}(p)$.

Proof: Note that the theorem is trivially true since we only add a new $t^{new}$ tuple into the queue if it does not exist previously. This guarantees that each invocation of the rule is unique.

A.3 Non-Linear Recursive Rules

Consider a rule with $n$ recursive predicates $p_1, p_2, \ldots, p_n$ and $m$ base predicates:

\begin{align*}
NL1: & \quad p_0 : = p_1, p_2, \ldots, p_n, b_1, b_2, \ldots, b_m \\
NL1-k: & \quad p_0^{new} : = \Delta p_1^{ed}, \ldots, p_n, b_1, b_2, \ldots, b_m
\end{align*}

We define the $k$th PSN rule as follows:

\begin{align*}
NL1-0: & \quad p_0^{new}^{new} : = \Delta p_1^{ed}, \ldots, p_n, b_1, b_2, \ldots, b_m
\end{align*}

In order for our proof of induction in Theorem 3 to work, we utilize a modified PSN, which produces the same results as PSN with extra derivations. In the modified PSN, if there are multiple derivations $t^i$ and $t^j$ for the same tuple $t$ marked with different iteration
numbers where \( j > i \), we ensure that the derivation with the lowest iteration number is eventually enqueued, in this example \( t' \). We use a modified PSN algorithm to ensure that, enqueuing \( t' \) even if a previous derivation with a greater iteration number exists. Note that this leads to multiple derivations but does not affect our proof of correctness.

**Claim 1** Consider the semi-naïve rule NLI-\( k \). \( \forall t' \in FP^t_{SN}(p) \), \( \exists t_j^1 \in FP^t_{SN}(p) \) and \( \exists t_j^1 \in FP^t_{SN}(p) \) s.t.
\[
t_0 : -t_1^i,t_2^i,...,t_m^i,b_1,b_2,...,b_m. \text{ Same for } FP^t_{SN}.
\]

**Theorem 3** \( \forall j (FP_{SN}(p_j) = FP_{PSN}(p_j)) \)

**Proof:** (By Induction). The base case \( \forall j (FP^t_{SN}(p_j) = FP^t_{PSN}(p_j)) \) is trivial since this is the initial set of input \( p_j \) tuples. Assume \( \forall j (FP^t_{SN}(p_j) = FP^t_{PSN}(p_j)) \) is true, we show that \( \forall j FP^t_{SN}(p_j) = FP^t_{PSN}(p_j) \) using the following two lemmas below.

**Lemma 1** \( \forall j (FP^t_{PSN}(p_j) \subseteq FP^t_{SN}(p_j)) \)

**Proof:** Consider a tuple \( t_0 \in FP^t_{SN}(p_0) \). From claim 1, \( t_0 \) is derived using rule:
\[
t_0 : -t_1^i,t_2^i,...,t_m^i,b_1,b_2,...,b_m. \text{ where } t_{j-1}^i \in FP^t_{SN}(p_j) \land t_0^i \notin FP^t_{SN}(p_0).
\]
Since we assume \( \forall j (FP^t_{SN}(p_j) = FP^t_{PSN}(p_j)) \), \( t_0 \notin FP^t_{SN}(p_0) \Rightarrow t_0^i \notin FP^t_{PSN}(p_j) \). Further, \( \forall j (t_{j-1}^i \in FP^t_{SN}(p_j)) \).

Based on the PSN algorithm, \( \forall j (t_{j-1}^i \in FP^t_{PSN}(p_j)) \) will be inserted into \( Q_j \). This will lead to the generation of \( t_0 \in FP^t_{PSN}(p) \).

**Lemma 2** \( \forall j (FP^t_{PSN}(p_j) \subseteq FP^t_{SN}(p_j)) \)

**Proof:** Consider a tuple \( t_0 \in FP^t_{SN}(p_0) \). From claim 1, \( t_0 \) is derived using rule:
\[
t_0 : -t_1^i,t_2^i,...,t_m^i,b_1,b_2,...,b_m. \text{ where } t_{j-1}^i \in FP^t_{SN}(p_j) \land t_0^i \notin FP^t_{SN}(p_0).
\]
Since we assume \( \forall j (FP^t_{SN}(p_j) = FP^t_{PSN}(p_j)) \), \( t_0 \notin FP^t_{SN}(p_0) \Rightarrow t_0^i \notin FP^t_{PSN}(p_j) \). Further, \( \forall j (t_{j-1}^i \in FP^t_{PSN}(p_j)) \). By executing rule \( t_0^i : -t_1^i,t_2^i,...,t_m^i,b_1,b_2,...,b_m. \) we generate \( t_0^i \in FP^t_{PSN}(p) \).

**Theorem 4** There are no duplicate evaluations in computing \( FP_{PSN}(p) \).

**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 \( ts(t_{k+1}) = \text{max}(ts(t_{k+1}^i), ts(t_{k+1}^i), ... , ts(t_{k+1}^i)) \) will be used to generate \( t_0^i \) at the inductive step, ensuring no duplicate evaluations.

**B. BURSTY NETWORK PROOFS**

**B.1 Derivation Tree**

A derivation tree for tuple \( t \) is defined as follows. The base tuples form the leaves of the tree. Each intermediate node in the tree is generated in a derivation step that consists of applying a rule with the children tuples as input. Consider a derivation step in which a given tuple \( t \) is generated by applying the rule \( t : -t_0,t_1,...,t_{m-1}. \) This derivation step is represented as a tree fragment, with the root set to \( t \), and \( \text{child}(t_j) = t_j. \) At the \( i^{th} \) iteration, a derivation tree of height \( i \) is generated.

**B.2 Centralize Updates**

Consider a series of updates to \( t \). We denote \( u(t,j) \) as the \( j^{th} \) update to \( t \). As shorthand denote \( t(0) \) denotes the initial value of \( t \), and \( t(j) \) denotes the \( j^{th} \) updated value of \( t \). For simplicity, we assume that deletes are modeled by setting a delete flag in \( t \), and considered an update.

**Claim 2** Let \( t \) be a derivation tree for \( t \) with \( n \) leaf nodes, \( l_1(t),l_2(t),...l_n(t) \) are applied to \( t \). These will be applied in the same order to other children, and in the absence of any changes to other children, the final result of \( t \) is computed using \( t : -l_0(t),l_1(t),...l_n(t) \). This FIFO ordering of updates per input predicate is guaranteed by the FIFO queues for each rule strand in the PSN algorithm.

Let \( t \) be the derivation tree for \( t \) with \( n \) leaf nodes, \( k^{th} \) leaf is \( l_k(t_0) \). Let \( t \) be another derivation tree for \( t \) with \( n \) leaf nodes, \( k^{th} \) leaf is \( l_k(t_0) \) for each of the leaf nodes.

**Theorem 5** \( \forall t (t(t) = t^{\prime}(t)) \)

**Proof:** (By Induction) Let \( t(t) \) be a tree of height \( i \). Consider derivation tree. At every step, make sure computing the latest value of children. Each leaf node has a latest-modified timestamp. Denote this as \( ts(leaf(t,k)) \).

**B.3 Distributed Updates**

**Claim 3** Any edge in derivation tree that requires communication means the child for edge is derived via a link-restricted rule.

**Assumption 1** Along any derived tuple derived via a link-restricted rule, there is FIFO along link.

**Theorem 7** For tuples derived using children derived using link-restricted rules, Theorem 6 holds.

**Proof:** From Claim 3 and Assumption 1, any tree edge with communication has to be FIFO. Hence, the same proof for Theorem 6 is valid here.

**B.4 Soft-State Proof**
Updates of base and derived tuples replace previous values (aka blind writes). So only the eventual copy matters. So timestamp of root is set accordingly.

C. REFERENCES