Customizable Routing with Declarative Queries

Boon Thau Loo* Joseph M. Hellerstein*† Ion Stoica*

ABSTRACT

To meet the demands of new Internet applications, recent work argues for giving end-hosts more control over routing. To achieve this goal, we propose the use of a recursive query language, which allows users to define their own routing protocols. Recursive queries can be used to express a large variety of route requests such as the k shortest paths, shortest paths that avoid (or include) a given set of nodes and least-loaded paths. We show that these queries can be efficiently implemented in the network, and in the simple case when all users request shortest paths, the communication overhead of our solution is similar to that incurred by a distance vector protocol. In addition, when only a subset of nodes issue the same query, the communication cost can be further lowered using automatic query optimization techniques – suggesting that declarative queries and automatic optimization are important in this domain. Finally, we outline the main challenges faced by our proposal, focusing on the expressiveness and efficiency of our proposal.

1. INTRODUCTION

In the current Internet architecture, the routing functionality is embedded in the infrastructure with end-hosts having little control over the path followed by their packets. This limits the ability of the infrastructure to evolve and meet the demands of new applications or provide new services. Several solutions have been proposed to address this problem. These solutions range from separating routing from the forwarding infrastructure [8], enabling end-hosts to choose their paths at the AS level [11], or even computing arbitrary routes [3].

We explore an approach in which end-hosts use declarative queries to express routing protocols. These protocols are then executed by the nodes in the routing infrastructure. In our examples, we use *Datalog*, a declarative query language targeted at the kind of *recursive* queries over graphs that are appropriate for routing [9].

Our use of a declarative query language is an attempt to achieve a sweet spot between expressiveness and security. Datalog offers more flexibility than most existing solutions in its ability to naturally express a large variety of routing protocols, as we demonstrate in Section 5. On the other hand, it is less general than running arbitrary code in Active Networks [3], but as a result can be more safely analyzed and executed (Section 2.2).

We also show that declarative queries need not hamper the efficiency of traditional protocols. For example, we show that in the simple case when all end-hosts issue the *same* Datalog query to find the shortest paths to other nodes, the communication cost to execute all these queries is roughly equal to the communication cost of a traditional distance-vector routing protocol. In addition, our simulation results demonstrate that when only a subset of nodes issue the same query, the communication cost can be further lowered using automatic query optimization techniques.

Finally, we observe that multiple alternative algorithms for route discovery have tradeoffs depending on constraints in the specification of the routing query, on the presence or absence of other queries in the network and on the network topology. This variability provides strong motivation for the use of declarative languages and runtime query optimization in routing protocols. We discuss the challenge of generating efficient query plans from the declarative queries in such a large distributed system. To support a large variety of concurrent customized routing protocols, we show how to exploit similarities across queries to share route computation. We also outline challenges and some suggested solutions for route stability.

2. SYSTEM MODEL

We model the routing infrastructure as a directed graph, in which each link is associated with a set of parameters (*e.g.* loss rate, available bandwidth, delay). The nodes in the routing infrastructure can either be IP routers or overlay nodes. Consider the example where an end-host A wishes to establish the best path to another end-host B based on a particular metric such as latency or bandwidth. A expresses its requirements as a declarative query in Datalog, and sends this query to one of the routing infrastructure nodes to which the host is directly connected. The result of this query can be either the entire path to B, in which case A uses source routing to send its packets to B, or the establishment of forwarding state at all routers from A to B, in which case A simply sends a packet to the first hop, with destination B.

Routing queries can be processed either in a centralized or distributed fashion. Centralized approaches [4] would require route providers to periodically gather network information from the infrastructure. Each routing query would then be sent to one or more of these providers, which would process the queries using their internal databases and return the result to the querier.

An alternative that we explore in this paper is to execute the query in the infrastructure in a distributed fashion. This alternative ensures that the routing infrastructure scales organically with the number of nodes, and adheres to the spirit of decentralization in the Internet itself. In this case, each infrastructure node runs a general-purpose recursive query processing engine instead of a traditional routing protocol. The query processing engine generalizes the operations performed by typical routing protocols, as we illustrate below.

2.1 Routing Information

To execute queries, each node maintains local infor-

*University of California at Berkeley
†Intel Research Berkeley

mation similar to the way a router maintains a routing table. This local information is directly accessible by the query processor. Initially, this local information consists of the properties associated with the node itself, and of the links to its neighbors. To keep with the terminology in databases, we will refer to local information as *base tuples*. Specifically, the format of the base tuples is as follows:

- **node(nodeID, ...)**. A *node* tuple stores information on a node in the network. The *nodeID* field is typically the routing address of the node. *nodeID* can be a logical address (such as Distributed Hash Table (DHT) [1] identifier) or a physical address (IP Address). Other fields representing node metrics (e.g. load) may also be included.
- **link(source, destination, ...)**. The routing table is represented as a set of *link* tuples, where a *link* tuple represents an edge from *source* to *destination*. Other fields representing link metrics (e.g. delay, loss rate, bandwidth) may also be included.

Each tuple is stored at the address indicated by the underlined address field. During query execution, the query processors generate intermediate data, called *derived tuples*. Derived tuples are specified by the query, and either stored locally or sent to a neighbor of the computing node for further processing. One example of a derived tuple we will see in our examples is a *path* tuple:

- **path(source,destination, pathVector, cost)**. A *path* tuple represents that *destination* can be reached from *source* along the path indicated by *pathVector*, where *cost* is the aggregate of all link costs along the path.

In addition to the base and derived tuples, a query processor generates *result tuples* that are part of the query answer. These tuples are either sent to the querier or stored in the network as forwarding state. Examples of result tuples in subsequent examples include *nextHop* and *shortestPath*.

2.2 Comparison with Active Networks

At one extreme, our proposal can be viewed as an instantiation of Active Networks: users write programs, and nodes in the network execute these programs. However, our proposal is more restrictive than traditional Active Network proposals. Datalog is a side-effect-free language, limited to polynomial time computations [7]. This restricts the potential for erroneous or malicious state modification and resource consumption. Like any query language, Datalog is logic-based and amenable to a range of static checking. Finally, our proposal is concerned only with processing on the control and not the data plane.

Despite these restrictions, we demonstrate in Section 5 that Datalog is sufficiently expressive for a large variety of routing protocols. At the same time, while these constraints make the challenges of achieving efficient execution and security more tractable, there still remain challenges to be tackled. We return to these issues in Section 4.

3. THE BASICS

We begin our discussion with the textbook example of a recursive query: the graph transitive closure, which can be used to compute network reachability. Using this example, we will introduce the syntax of Datalog, show the generation of a query plan from Datalog, and step through the communication patterns of running the query plan within a network. Last, we show that the execution of the query resembles the well-known path vector or distance vector routing protocols.

3.1 Datalog Program Syntax

Datalog is similar to Prolog, but hews closer to the spirit of declarative queries, exposing no imperative control. Each Datalog *program* consists of a set of declarative *rules* and *queries*. Following the Prolog-like conventions used in [9], names for tuples, predicates, function symbols and constants begin with a lower-case letter, while variables names begin with an upper-case letter. A Datalog *rule* has the form <*head*> :- <*body*>, where the body is a list of predicates over constants and variables, and the head defines a set of tuples derived by variable assignments satisfying the body's predicates. A *query* is just a specific rule of interest as output. A Datalog *program* consists of a set of rules and a query; typically the rules reference each other in a cyclic fashion to express recursion. Presented with a program, a Datalog system will find all possible assignments of tuples to unbound variables in the query that satisfy the rules in the program.

Our first example, the *Network-Reachability* program, takes as input link tuples, and computes the set of all paths (represented by path tuples). In all our examples, *S*, *D*, *C* and *P* abbreviate the *source*, *destination*, *cost* and *pathVector* fields respectively for both the link and path tuples. As before, the address fields indicating the location of the tuples are underlined. We begin our discussion by looking only at the part of the query written in **bold** text, ignoring the rest of the text for a moment.

NR1: **path(S,D**,P,C) :- **link(S,D**,C),
P = concatPath(link(S,D,C), nil).

NR2: **path(S,D**,P,C) :- **link(S,Z**,C_1),
path(Z,D,P_2,C_2), C = C_1 + C_2,
P = concatPath(link(S,Z,C_1),P_2).

Query: **path(S,D**,P,C).

The above program works as follows. Rule NR1 produces new one-hop paths from existing link tuples, storing them at the source node. Rule NR2 recursively produces path tuples of increasing length by matching the destination fields of existing links to the source fields of previously computed paths; the new path tuples are stored the source node.

The query does not impose a restriction on either source or destination as both *S* and *D* are unbound variables. Hence, the program computes the *full transitive closure* containing path tuples between all possible pairs of reachable nodes. If the program is only interested in the paths for node *b*, then the query would be *path(b,D,P,C)*, with the source field bound to constant *b*.

We now focus on the remaining portions of rules NR1 and NR2. The expression $P = concatPath(L, P_1)$ is a predicate that is satisfied if P is the path vector produced by prepending link L to the existing path vector P_1. With these additions, rules NR1 and NR2 also compute the total path costs, and the path vectors themselves.

3.2 Query Plan Generation

Figure 1 shows a query "plan" for the Datalog program. A query plan is a dataflow diagram consisting of

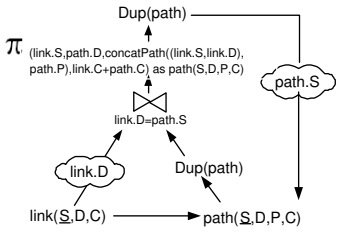

Figure 1: ***Query Execution Plan for the Network-Reachability Program.***

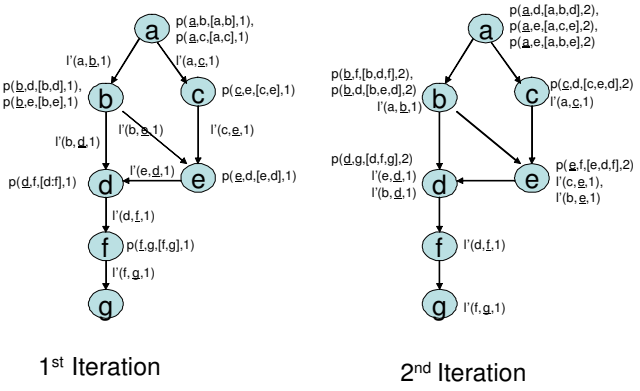

Figure 2: ***Nodes in the network are running the query plan in Figure 1. p(S,D,P,C) abbreviates path(S,D,P,C) and l(S,D,C) abbreviates link(S,D,C). Link costs in our example are set to 1, and hence path cost is equal to the number of hops. l'(S,D,C) refers to link tuples that are shipped and cached at the destination nodes. Only new path tuples generated at each iteration are shown.***

relational operators and arrows indicating the flow of tuples. The transformation to this query plan is as follows. Rule NR1 is a simple renaming of existing link tuples to path tuples, and this is shown by the rightward arrow from *link(S,D,C)* to *path(S,D,P,C)*.

Rule NR2 requires a relational join operator to match the destination fields of link tuples (*link.D*) with the source fields of existing path tuples (*path.S*). The fields used for matching are a result of variable unification of the common *Z* variables in rule NR2. The join operator, represented by the $\bowtie$ symbol, matches link and path tuples from the inputs on the appropriate attributes. The projection operator, represented by the π symbol, takes as input the output of the join and a list of fields, extracts and renames only the listed fields to form its output tuples. The *Dup* operator removes duplicate tuples from its input stream. Note that unlike many textbook query plans, the dataflow here forms a cycle, which captures the recursive use of the *path* rule definition in the query.

The clouds in the figure are required only when the query plan is executed in a distributed fashion. They represent the forwarding of tuples from one node to another, and are labeled with the destination node. The first cloud (*link.D*) ships link tuples to the 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*) ships new *path* tuples computed from the join back to their source nodes for further processing.

3.3 Query Plan Execution

When the query plan is executed, the flow of tuples in the network enables nodes to exchange the routing information necessary to compute the queried routes. Figure 2 shows the resulting communication pattern for executing the query plan in Figure 1 on *all* nodes in the network. Our example is based on a directed graph, although this discussion applies to both directed and undirected graphs.

We will describe the communication in stages, where each stage or *iteration* represents a "round of communication", in which all nodes exchange tuples from the previous iteration. Each iteration represents the traversal of a "cloud" in Figure 1. The first iteration derives single-hop path tuples from the first rule of the program. It does this by traversing the *link.D* cloud, which ships link tuples to the address in their destination field, where they are cached for the duration of the query. Because the query has no recursion on the *link* table, all subsequent iterations involves the other cloud (*path.S*). In the 2^{nd} iteration, the shipped link tuples are joined with existing one-hop path tuples to produce two-hop path tuples. These tuples are then sent back to the source nodes (the *path.S* cloud) and three-hop path tuples are computed. The query is completed after k iterations, where k is the diameter of the network.

3.4 Path Vector or Distance Vector Protocol

The computation of the above query resembles the computation of the routing table in a path vector or distance vector protocol. The computation starts with the source computing its initial reachable set (which consists of all neighbors of the source) and shipping it to all its neighbors. In turn, each neighbor updates the reachable set with its own neighborhood set, and then forwards the resulting reachable set to its own neighbors. The distance vector computation can be expressed with minor modifications to our previous program (modifications in **bold**):

DV1: path(S,D,**D**,C) :- link(S,D,C),
P = concatPath(link(S,D,C), nil).

DV2: path(S,D,**Z**,C) :- link(S,Z,C_2),
path(Z,D,**W**,C_1), C = C_1 + C_2.

DV3: **shortestLength(S,D,*min*<C>) :- path(S,D,Z,C)**

DV4: **nextHop(S,D,Z,C) :- path(S,D,Z,C), shortestLength(S,D,C)**

Query: nextHop(S,D,Z,C).

Aggregate constructs are represented as functions with arguments within angle brackets (<>). DV1 and DV2 are modified from the original rules NR1 and NR2 to ensure that the path tuple maintains only the next hop on the path, rather than the entire path vector itself. DV3 and DV4 are added to set up the routing state *nextHop(S,D,Z,C)* stored at node *S*, where *Z* is the next hop on the shortest path to node *D*. The above query can still be further optimized (Section 6). The main difference between this query and the actual distance vector computation is that rather than sending individual path tuples between neighbors, the traditional distance vector method batches together a vector of costs for all neighbors.

4. CHALLENGES

Based on the *Network-Reachability* example, we have

identified several challenges that we need to address before our proposal becomes feasible:

Expressiveness: How expressive and flexible is the Datalog language in expressing various routing policies? What are the limitations of this language?

Efficiency: Can Datalog queries be executed efficiently in a distributed system? It appears that the answer to this question hinges on two sub-questions. Can plan generation techniques be adapted or developed to enable Datalog queries to perform well in a large network system? And, given that there will be many queries issued concurrently, how can we reuse the work done by previous or concurrent queries to reduce redundant work?

Stability and Robustness: Given that the network is dynamic and the resulting routes can be used for a long time, how can we efficiently maintain the robustness and accuracy of routes?

Security: Is Datalog an acceptably safe language to expose in the network infrastructure? Compared to Active Networks, our proposal trades expressivity for better security opportunities. Finding the right balance between expressiveness and security is an open problem.

In the rest of the paper, we address the first two challenges, focusing on the *expressiveness* and *efficiency* of our proposal. We also outline solutions and open issues related to stability and robustness.

5. EXPRESSIVENESS

In its purest form, Datalog has the ability to express most polynomial-time computations [7]. Most implementations of Datalog enhance it with a limited set of function calls, including boolean predicates, arithmetic computations and simple string manipulation (e.g. the *concatPath* function). To illustrate the flexibility of Datalog, we provide several examples of useful routing protocols. To demonstrate the ease of use, we present the examples incrementally, starting from the base rules NR1 and NR2 from *Network-Reachability* example in Section 3, and adding simple modifications to create new routing protocols.

5.1 Best-Path Routing

By adding two rules BPR1 and BPR2, the following *Best-Path* program computes the best path for the path metric *C*:

#include(NR1,NR2)

BPR1: bestPathCost(S,D,AGG<C>) :- path(S,D,P,C).

BPR2: bestPath(S,D,P,C) :- bestPathCost(S,D,C), path(S,D,P,C).

Query: bestPath(S,D,P,C).

#include is a macro used to include earlier rules. We have left the aggregation function (*AGG*) unspecified. By changing *AGG* and the way that the path cost *C* is computed, the *Best-Path* program can generate best paths based on any metric, such as average link cost, least total aggregate node load, average link bandwidth, etc. For example, if the query is used for computing the shortest paths, *min* is the appropriate replacement for *AGG* in rule BPR1. The resulting *bestPath* tuples are stored at the source nodes, and are used by end-hosts to perform source routing. The two added rules BPR1 and BPR2 do not result in extra messages being sent beyond those generated by rules NR1 and NR2. This is because path tuples computed by rules NR1 and NR2 are stored at the source nodes, and *bestPathCost* and *bestPath* tuples are generated locally at those nodes. Instead of computing the best path between any two nodes, we can also compute *any* path or the *Best-k* paths between any two nodes.

5.2 Policy-Based Routing

Each individual node can customize its own local rules that represent policy decisions within its own routing domain. For example, certain nodes may refuse to carry traffic for some other nodes. We can express this kind of policy constraint by adding an additional rule:

#include(NR1, NR2)

PBR1: permitPath(S,D,P,C) :- path(S,D,P,C), **excludeNode(S,W), ¬inPath(P,W).**

Query: permitPath(S,D,P,C).

In this program, *excludeNode(S,W)* is a tuple that represents the fact that node *S* does not carry any traffic for node *W*. The program includes a function $inPath(P, W)$ that returns true if node *W* is along the path vector *P*. If BPR1 and BPR2 are included as rules, we can generate *Best-Path* that meets the above policy.

5.3 Dynamic Source Routing

All of our previous examples use what is called *right* recursion, since the recursive use of *path* in the rule (NR2, DV2) appears to the right of the matching *link*. The query semantics do not change if we flip the order of *path* and *link* in the body of these rules, but the execution strategy does change. In fact, using *left recursion* as follows, we achieve the Dynamic Source Routing (DSR) protocol [6]:

#include(NR1)

DSR1: path(S,D,P,C) :- **path(S,Z,P_1,C_1), link(Z,D,C_2),** P = concatPath(P_1, link(Z,D,C_2)), C = C_1 + C_2.

Query: path(N,M,P,C).

5.4 Disjoint Paths Routing

One limitation of Datalog is the inability to express the *Best-k-Disjoint* paths between two specified nodes. This problem is known to be NP-complete. A heuristic approach greedily generates one disjoint path at a time, keeping track of previously discovered nodes *N* from source *S* as *avoidNodes(S,N)* tuples. To illustrate, the following program computes *k* (hopefully good) node-disjoint paths from node *a* to node *b*:

#include(BPR1, BPR2)

DPR3: path(S,D,P,C) :- link(S,D,C), P=concatPath((link(S,D,C), nil), **¬avoidNodes(S,D).**

DPR4: path(S,D,P,C) :- link(S,Z,C_1), path(Z,D,P_2, C_2), P=concatPath(link(S,Z,C_1), P_2), C=C_1+C_2, **¬avoidNodes(S,Z).**

DPR5: **avoidNodes(S,N) :- node(N), bestPath(S,D,P,C), inPath(P,N), N≠S, N≠D.**

Query: bestPath(a,b,P,C)

Each invocation of the program produces a bestPath tuple. The nodes along the pathVector field for the newly produced bestPath tuple are added as *avoidNodes* derived tuples, and the program is executed up to *k* times to get *k* disjoint paths.

6. EFFICIENCY

In this section, we address the challenge of generating efficient query plans from the declarative queries. We utilize three well-known query optimization techniques used in centralized deductive database systems, and discuss how useful they will be in generating efficient query plans in our distributed environment. They are *aggregate selections*, *magic sets rewriting* and *left-right recursion rewriting*. In addition, we address previously unexplored challenges introduced by our environment, which requires work-sharing among a diverse set of queries in order for the routing infrastructure to scale.

6.1 Pruning Unnecessary Paths

A naive execution of queries with aggregates such as *Shortest-Path* and *Least-Loaded-Path* requires the enumeration of all possible paths. This can be avoided with *aggregate selections* [10]. To illustrate, in Figure 2, there are two different paths from node b to node f, but only the shorter of the two is required when computing shortest paths. By maintaining a "min-so-far" aggregate value for the current shortest path cost from node b, we can avoid sending path tuples to neighbors if we know they cannot result in the shortest path. Such pruning based on running aggregates only works for monotonic functions like min or max. Aggregate selections result in significant savings for dense networks, where there can be many paths between any two nodes.

6.2 Limited Sources and Destinations

The *Network-Reachability* example in Section 3 requires all nodes to participate in the query plan. This is overkill when only a subset of nodes want to know their reachable set. A program rewrite technique called *magic sets rewriting* [2] can be used to limit query computation to only a portion of the graph, based on the nodes issuing the query. For example, if nodes b and e are the only nodes issuing the path query, the rewritten example is as follows:

MRR1: **magicSources(D) :- magicSources(S), link(S,D,C).**
MRR2: path(S,D,P,C) :- **magicSources(S)**, link(S,D,C),
P = concatPath(link(S,D,C), nil).
MRR3: path(S,D,P,C) :- **magicSources(S)**, link(S,Z,C_1),
path(Z,Y,P_2,C_2), C = C_1 + C_2,
P = concatPath(link(S,Z,C_1),P_2).
MRR4: **magicSources(b).**
MRR5: **magicSources(e).**
Query: path(N,M,P,C).

As before, modifications indicated in **bold** are made to rules NR1 and NR2. After the rewrite, only nodes reachable from b and e need to participate in this query. The program can be further optimized by combining common sub-rules in MRR1, MRR2 and MRR3. Magic sets can also be used to limit computations by destinations, or by both source and destination nodes concurrently.

6.3 Left-Right Recursion Rewrite

In Figure 2, suppose node b is the only source node. Even with the use of magic sets, the paths for nodes g, f and d are computed before source node b can compute its paths. We can avoid this extra computation by rewriting the program using *left* recursion:

#include(MRR2, MRR4, MRR5)
MLR1: path(S,D,P,C) :- magicSources(S), **path(S,Z,P_1,C_1),**
link(Z,D,C_2), C = C_1 + C_2.
P = concatPath(P_1, link(Z,D,C_2)).
Query: path(N,M,P,C).

As pointed out in Section 5, executing the program in a left-linear fashion bears close resemblance to dynamic source routing. This approach reduces communication overhead by computing only the required paths for the source nodes b and e. Each node computes new path tuples by recursively following the links along all reachable paths, and does not depend on paths generated by neighboring nodes. Hence, the main drawback of this approach is that source nodes do not share the paths computed among themselves even when the paths overlap. This leads to redundant work as the number of source nodes increases. The redundancy may be more apparent for dense networks since there will be more overlapping paths among different source nodes. In general, one would like an optimizer to automatically choose whether to use left or right recursion (or, more generally, the order of predicate evaluation in the rules).

6.4 Multi-Query Sharing

A key requirement for scalability is the ability to share route computation among a potentially large number of queries. If all nodes are running the same query, using right-recursion ensures that each node directly utilizes path information sent by neighboring nodes, hence achieving 100% sharing.

On the other hand, if a small subset of nodes are issuing the same query, using left-recursion achieves lower message overhead as we will see in Section 8. To facilitate sharing among nodes issuing the same query, previously computed tuples are reused whenever possible. For example, revisiting the example network in Figure 2, if node d's computed path tuples are materialized (computed and stored) and stored in the network, they can be reused by both nodes b and e, and hence avoid multiple traversals of the path $d \rightarrow f \rightarrow g$. Further sharing is achieved if the resulting path tuples are sent back via the reverse path back to the source node to be reused by other queries. For example, when node a computes its shortest path to node g, the nodes on the reverse path (nodes b, d, f and g) can cache information on the shortest path (and sub-paths) to node g, to be reused by subsequent queries.

7. STABILITY AND ROBUSTNESS

In practice, we expect the computed routes to be used for a long time. These routes may be invalidated as the underlying network changes. Some basic mechanisms are available to handle this issue effectively; we sketch an approach here.

Each base tuple should be maintained as soft-state with an associated timeout. A smaller timeout ensures fresher routes at the expense of more messaging overhead. The base tuples are periodically renewed with new values, or deleted when expired. Derived tuples computed using base tuples should be timestamped based on the oldest base tuple used in the computation, so that they expire when any of their components expire.

To ensure that new tuples trigger only incremental recomputations, we need not discard the state of a query after a route is established. Instead, the intermediate state of each query can be retained in the network until the query is no longer required. This intermediate state

includes any shipped tuples used in join computation, and any intermediate derived tuples. This state is deleted at the end of the query, or whenever the base tuples expire, whichever is earlier. With a bit of subtlety in the query processing algorithms, the insertion of a new base tuple should only trigger a minimal incremental computation to update the current state of query execution.

Beyond minimizing overhead of incremental route maintenance, there are other challenges related to route flapping, and unwanted interactions between diverse queries with conflicting metrics. We plan to investigate these issues in future work.

8. PRELIMINARY EVALUATION

We present a preliminary evaluation via simulation. The simulation computes new tuples based on the query workload, and identifies those tuples that need to be shipped at each iteration during query execution. The input network is a connected undirected graph of size 1000, with 3000 random links. The input query is the shortest path query with aggregate selections and magic sets rewrite. Our experiments vary the number of nodes issuing the queries, and count the number of messages (tuples) incurred during the execution of the queries. In all our experiments, the baseline uses right recursion to execute the query and is labeled as *Right-Full* in all graphs. *Right-Full* computes all-pairs shortest paths regardless of the number of nodes issuing the query, and resembles the computation of the routing table in a path vector or distance vector protocol.

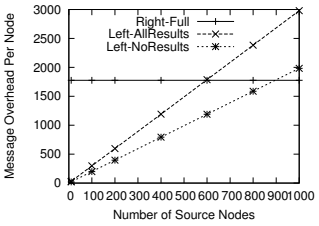

Figure 3: ***Message Overheads Per Node***

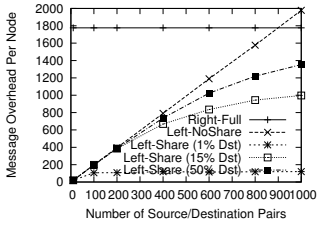

Figure 4: ***Message Overheads Per Node with Sharing***

In Figure 3, the *Right-Full* baseline incurs a per-node message overhead of 1800 messages. *Left-AllResults* and *Left-NoResults* show the per-node message overhead of using left recursion in query execution, where each source query node computes its own shortest-paths independently of other nodes. *Left-AllResults* includes the cost of returning all computed shortest-path results, while *Left-NoResults* does not include the cost of returning any results; The pair is used to provide an upper and lower bound respectively when the number of destinations specified in the query varies from 0 to 1000. Our experimental result shows that when there are few sources and destination nodes involved in the queries, the message overhead can be much lower compared than computing all-pairs shortest paths. However, as the number of source nodes increases, the message overhead increases linearly, even exceeding *Right-Full* at 600 and 900 source nodes respectively due to the lack of sharing.

In Figure 4, we examine the effects of work-sharing to reduce redundant computation when using left recursion. Here, we limit each query to a source and destination pair. *Left-NoShare* and *Left-Share* show the message overhead of left recursion without and with sharing. Sharing occurs when query results are cached on the reverse path as described in Section 6.4 for use by subsequent queries. When the percentage of destination nodes involved in the queries is low (1% Dst), the cache hit rate is high. As the percentage of destination nodes increases (50% Dst), the cache hit rate is lowered, and hence less sharing is achieved.

Our simulation results demonstrate that the optimal query plan is affected by the number of nodes issuing the same query. Other factors such as the presence of different queries with correlated metrics and the network topology may also affect the choice of query plan. Given that the query workload and network are dynamic and may not be known a priori, we intend to explore adaptive query optimization techniques that results in an optimal query plan at run time.

9. CONCLUSION

In this paper, we argue that recursive queries are flexible enough to express a large variety of routing protocols easily. We also argue that they allow for efficient execution, are amenable to automatic optimization, and they offer an attractive mixture of expressiveness and safety. Many research questions are open, including a study of various query execution strategies, automatic query optimization, automatic sharing across queries, techniques to tolerate flux in the topology, and of static analysis techniques to ensure that queries can be executed safely. To ground our investigation, we have added recursive query features to the PIER [5] DHT-based query engine, and are using it to build a customizable routing infrastructure.

10. REFERENCES

[1] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, February 2003.

[2] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *PODS*, 1987.

[3] D. Tennenhouse and J. Smith and W. Sincoskie and D. Wetherall and G. Minden. A Survey of Active Network Research. In *IEEE Communications Magazine*, 1997.

[4] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The Case for Separating Routing From Routers. In *FDNA*, 2004.

[5] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, Sep 2003.

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

[7] Phokion Kolaitis and Moshe Vardi. On the Expressive Power of Datalog: Tools and a Case Study. In *PODS*, 1990.

[8] Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. Routing as a Service. Technical Report UCB-CS-04-1327, UC Berkeley, 2004.

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

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

[11] Xiaowei Yang. NIRA: A New Internet Routing Architecture. In *Proceedings of FDNA-03*, 2003.