Analyzing P2P Overlays with Recursive Queries

Boon Thau Loo, Ryan Huebsch, Joseph M. Hellerstein, Timothy Roscoe, and Ion Stoica

IRB-TR-03-045

November, 2003

DISCLAIMER: THIS DOCUMENT IS PROVIDED TO YOU "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE. INTEL AND THE AUTHORS OF THIS DOCUMENT DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS, RELATING TO USE OR IMPLEMENTATION OF INFORMATION IN THIS DOCUMENT. THE PROVISION OF THIS DOCUMENT TO YOU DOES NOT PROVIDE YOU WITH ANY LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS

Intel Research Berkeley

Analyzing P2P Overlays with Recursive Queries

Boon Thau Loo[*], Ryan Huebsch[*], Joseph M. Hellerstein[*†], Timothy Roscoe[†], and Ion Stoica[*]
[*]*University of California at Berkeley*, [†]*Intel Research Berkeley*

Abstract

We explore the utility and execution of recursive queries as an interface for querying distributed network graph structures. To illustrate the power of recursive queries, we give several examples of computing structural properties of a P2P network such as reachability and resilience. To demonstrate the feasibility of our proposal, we sketch execution strategies for these queries using PIER, a P2P relational query processor over Distributed Hash Tables (DHTs). Finally, we discuss the relationship between in-network query processing and distance-vector like routing protocols.

1 Introduction

Much of the state of the Internet, and the applications running on top of it, is captured in graph structures, ranging from physical links, routing tables, multicast trees, hypertext structures, and peer-to-peer link graphs. Processing of information structured as graphs is a significant part of the problem of monitoring and managing such systems.

In this paper, we argue for the use of *recursive queries* as a powerful tool for understanding and controlling structural properties of peer-to-peer and overlay networks. Recursive queries allow a query result to be defined in terms of itself. This is particularly useful for querying relationships that themselves exhibit recursive structure, such as the reachability relationship of a network graph.

We consider two possible settings for realizing recursive query functionality: *embedded network queries* and *external network queries*. With embedded network queries, we assume that each node in the network embeds query functionality. We then show how users can leverage this functionality to compute a number of useful examples: (1) the set of nodes reachable from a given node, (2) the shortest path between two nodes, and (3) the number of paths between two nodes. Furthermore, we show how the reachability query can be efficiently executed in a DHT network with embedded query support. In contrast, with external network queries we assume that the query is executed on a *separate* DHT-based query infrastructure such as PIER. As an application, we show how this architecture can be used to perform a distributed crawl of Gnutella. This is just one example of using a distributed query processor to monitor an existing distributed graph; another example is a distributed hypertext crawler.

1.1 Context

A number of recent efforts have produced declarative query engines for monitoring and extracting information from networks [1, 6, 8, 11, 13, 14]. The common goal of all these systems is to provide a high-level, declarative interface for extracting information about network characteristics, without requiring a programmers to worry about the details of performing distributed queries efficiently. Our work continues in this spirit. While some of the prior research has focused on executing certain queries efficiently "in-network", none has focused on querying the structure of the network graphs themselves.

Declarative queries on graphs can be achieved only with recursive queries, which were a topic of intense research in database theory circles in the 1980's and early '90's. However, the utility of recursive queries in centralized databases has traditionally been dismissed by the database systems community: a senior researcher reiterated as recently as 1998 that "no practical applications of recursive query theory have been found to date" [7]. We argue here that recursive queries have great practical value as a declarative interface to multi-hop networks – both their native topologies, and the graphs overlaid upon them. This requires more than a revival of 1980's database theory, however. Unlike traditional deductive databases, network graphs are large, distributed, dynamic, and often based on soft state. These properties present new, practically grounded research challenges.

It is interesting to note that the computation of a recursive query resembles the computation of a routing table by a distance-vector protocol: the node initiating the query sends a set of facts to its neighbors, which in turn updates this set based on their local information and then propagate the resulting set further (see Section 5). In this way, recursive query processing can be seen as a generalization of existing distance vector like protocols. This suggests using recursive queries as a generic substrate for developing more flexible and powerful routing protocols.

2 Datalog

We give our examples using *Datalog* [10] programs, where each program consists of a set of rules and queries. In this paper, we focus on programs with recursion, although Datalog can of course be used to express nonrecursive programs as well. Datalog is similar to Prolog, but hews closer to the spirit of declarative queries, and exposes no imperative control (either explicitly or implicitly). Following the Prolog-like conventions used in [10], names for predicates, function symbols and constants begin with a lower-case letter, while variables names begin with an upper-case letter. Aggregate constructs are represented as functions with arguments within angle brackets (<>). Data, or *facts*, are conventionally represented by predicates with constant arguments; these can be thought of as database tuples. Presented with a query, the system will attempt to find a complete set of variable bindings to satisfy the rules, and return the values requested in the query.

3 Embedded Network Queries

In this section, we consider a generic network architecture in which each node embeds query processing functionality. Next, we give some examples in which users can leverage this functionality to compute various network properties. While these examples are straightforward, they serve the purpose to demonstrate the expressive power of the recursive declarative queries.

3.1 Reachability

The textbook example of recursive query is graph transitive closure, which can be used to compute network reachability. We assume the query processor at node *X* has access to *X*'s routing table. Let *link(X,Y)* denote the link between node *X* and its neighbor *Y*. Then, the following simple program computes the set of all nodes reachable from node *a*.

R1: *reachable(X,Y) :- link(X,Y).*
R2: *reachable(X,Y) :- link(X,Z), reachable(Z,Y).*
Query: *?- reachable(a,N).*

There are many useful enhancements of this program. One is to change the query to simultaneously compute all *reachable* pairs, not just those starting at *a*. Another is to compute reachable pairs that are within a given *hop count* from the initial node. A third possibility is to extend the program to check for cycles between any two nodes in the network.

3.2 Shortest Path

Suppose now that we have a cost *C* associated with each *link(X,Y,C)*. Then, the following program computes the shortest path from nodes *a* to *b* and its cost:

R3: *shortestPath(X,Y,P,C) :- shortestLength(X,Y,C),*
path(X,Y,P,C).

R4: *path(X,Y,P,C) :- link(X,Z,C_2),*
path(Z,Y,P_1,C_1),
P = addLink(link(X,Z),P_1),
$C = C_1 + C_2$.

R5: *path(X,Y,P,C) :- link(X,Y,C),*
P = addLink(link(X,Y), nil).

R6: *shortestLength(X,Y,min<C>) :- path(X,Y,P,C).*

Query: *?- shortestPath(a,b,P,C).*

The expression *P = addLink(L, P_1)* is satisfied if *P* is the path produced by appending link *L* to the existing path P_1. As it stands, this query is inefficient since it enumerates all possible paths. However, query optimization techniques exist to improve performance by automated rewriting of the query [12]. We omit examples of rewritten queries here for space reasons.

As with reachability, this query can be easily modified to request all-pairs shortest paths, the shortest of all paths that have some particular property, etc.

3.3 Routing Resilience

Another query of interest computes the number of paths between any two nodes, which provides a measure of the routing resilience. Given two nodes *a* and *b*, this program computes the number of paths between them:

R7: *path(X,Y,P) :- link(X,Z),*
path(Z,Y,P_1),
P = addLink(link(X,Z),P_1).

R8: *path(X,Y,P) :- link(X,Y),*
P = addLink(link(X,Y), nil).

R9: *numberPaths(X,Y,count<P>) :- path(X,Y,P).*

Query: *?- numberPaths(a,b,N).*

This query can be extended straightforwardly to generate the number of disjoint paths between two nodes.

3.4 *Cast Trees

Another class of useful queries are those that examine trees constructed within the network. Such trees are often used for multicast or in-cast (aggregation). A number of such queries are possible. For example, a query could find the height of each subtree and store the height value at the root of the subtree. In another example, a query could find the *imbalance* in the tree – the difference in height between the lowest and highest leaf node. Yet in another example, a query could identify all the nodes at a given level of the tree (referred to as the "same generation" query in the Datalog literature). We omit the actual programs for these examples in the interests of brevity.

4 External Network Queries

Recursive queries can also be distributed in a DHT to query the structure of a different network in a parallel fashion. In this section, we will show how PIER can be used to perform a crawl of Gnutella using recursion. The "raw data" gathered from the crawl can be used as input to other recursive queries to monitor the Gnutella network. Such queries can be used to study file-sharing workloads, and potentially to improve the search capabilities of Gnutella as well. Our examples can easily be adapted to other distributed graphs, e.g., to build decentralized web crawlers or crawl other P2P networks besides Gnutella. The only requirement on these networks is that their nodes can be queried for link information, as with Gnutella and the Web.

4.1 Crawling Gnutella

The following program performs a crawl of Gnutella, and serves as a basis for more complex queries, such as indexing the content served by the network. It uses two relations, *node(X)* means that X is a Gnutella node, and *link(X,Y)* means there is an existing Gnutella neighbor link from node X to node Y. For simplicity, we assume that links are directed.

```
R1:    node(X,0) :- startset(X).
R2:    link(X,Y) :- node(X,Hop),
                              gnutellaPing(X,Y).
R3:    node(Y,Hop) :- node(X,Hop-1),
                              link(X,Y),
                              Hop < k.
Query:  ?- node(N,D).
```

Given a set *startset* of initial nodes (represented as IP addresses), this query returns the set of Gnutella nodes within *k* hops of the startset, together with their distance in hops. Rule 1 initializes the *node* relation. Rule 2 expands the *link* set using the predicate *gnutellaPing(X,Y)*, which is satisfied if *X* believes *Y* is a neighbor of it; this predicate is evaluated by making a connection to the Gnutella node *X* and requesting its neighbors. Rule 3 expands the *node* set by joining nodes to links as long as the nodes are still within *k* hops of the startset. The query returns such all (node, distance) pairs. If *k* is set to infinity, this results in an exhaustive crawl of the Gnutella network.

4.2 Distributing the Crawl

The query is executed in a distributed fashion as follows. Each *crawler node* which runs the PIER query engine is responsible for crawling a different set of nodes in the Gnutella network. The query is first sent to all the crawler nodes, and set to run for a predetermined duration. The start set is partitioned (rehashed/published) among the participating crawler nodes by assigning each Gnutella node to the DHT node that is the "owner" of the hash of its IP address. This starts a distributed, recursive computation in which the query continuously scans for new nodes to be crawled. As each new node is discovered, it is similarly rehashed on its IP address to a crawler node, which continues the crawl from there.

There are good reasons to distribute the crawl. A naive centralized crawler may be unable to capture an accurate snapshot of the Gnutella network, as the crawl would have to run over a long period of time to avoid saturating the incoming bandwidth to the crawler site, especially if the query is expanded to return additional data about the crawled nodes.

A distributed crawler avoids this problem by balancing the bandwidth consumption across links. With a distributed set of nodes, we can also potentially exploit geographic proximity in the crawl by assigning each Gnutella node to be crawled by a "close" crawler node, where "closeness" is defined in terms of network latency, rather than partition by IP addresses.

Note that the distributed query processor naturally coordinates and partitions the work across the participating nodes. Using only data partitioning of the intermediate tables, it manages to parallelize the crawl without any explicit code to ensure that multiple sites do not redundantly crawl the same links.

4.3 Other Queries

The link and node information gathered in the crawl program can be published on the fly back to the DHT for further processing. Given rules to derive the link information, we can pose additional interesting network queries such as the ones discussed in Section 3.

In addition to querying the graph topology itself, recursive queries can be used to query summaries of particular nodes within its *horizon* (nodes that are reachable within a bounded number of hops). Horizon summary queries that we can compute include the number of files shared by all nodes within the horizon, the number of free-loaders within the horizon, the average number of files stored per node, the most popular files in the horizon, and so on.

5 Embedded Network Queries over DHTs

In this section, we discuss how recursive queries are processed over a DHT using PIER, and examine the communication patterns that result from publishing derived Datalog facts into the DHT. Note that DHT publishing is the only source of communication during query execution; no explicit messaging is used during execution. As our example, we use the program described in Section 3.1 that computes the set of *reachable* nodes from a given node:

```
R1:    reachable(X,Y) :- link(X,Y).
R2:    reachable(X,Y) :- link(X,Z), reachable(Z,Y).
```

Assume we want to simultaneously compute the set of reachable nodes from node *a*, and the set of reachable nodes from node *b*, respectively. We will show how these two queries can be efficiently computed together.

Figure 2 shows the query execution plan for computing reachability information beyond the first hop. The execution is based on Datalog's bottom-up ("forward-chaining") mechanism, which starts by applying the rules in the program to all existing *link* facts. The Clouds in the figure represent rehashing (publishing) tuples into the DHT and are labeled with the keys used to rehash them. The communication patterns during query execution are shown in Figure 1. We omit the computation for links involving *g* and *f* for clarity; we will later show that these links are in fact irrelevant to the computation.

The computation of a reachability query resembles the computation of the routing table in a distance vector protocol. The computation starts with the source computing its initial reachable set (which consists of all neighbors of the source) and publishing it to all its neighbors. In turn, each neighbor updates the reachable set with its own

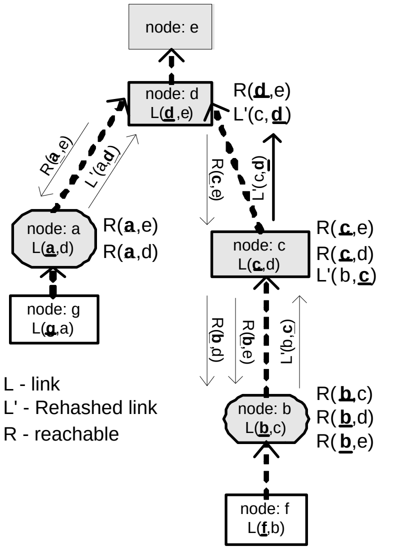

Figure 1: *Thick dotted lines are DHT overlay links. Thin solid lines are communication messages during query execution.*

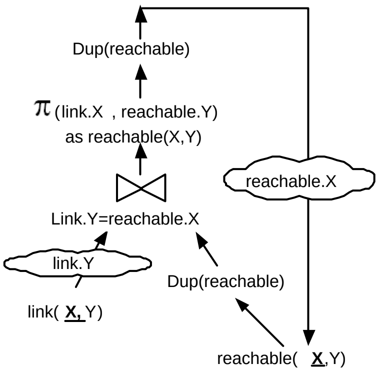

Figure 2: *Query Execution Plan for the Reachable Program.*

neighborhood set, and then forwards the resulting reachable set to its own neighbors.

To illustrate further, we step through the communication necessary for the computing *R(b,e)* for node *b*.

1. *b* rehashes *L(b,c)* to *c*.
2. *c* rehashes *L(c,d)* to *d*.
3. *d* joins *L(c,d)* and *R(d,e)* and rehashes result *R(c,e)* to *c*.
4. *c* joins *L(b,c)* and *R(c,e)* and rehashes result *R(b,e)* to *b*.

We make three observation:

Natural Communication Patterns: The communication patterns described above follow naturally those of a distance vector routing protocol computing reachability information in a multi-hop, decentralized fashion.

Work Sharing: Nodes *a* and *b* share a common reachable node *d*. We only need to compute *reachable* facts sourced at node *d* once. For example, *R(d,e)* would be computed once and stored at node *d*, and subsequently used to compute *R(a,e)* and *R(b,e)*. Such storage and sharing of intermediate results is known in deductive databases as work *memoization*.

Irrelevant Facts: In our example, links *L(g,a)* and *L(f,b)* are not needed, but are used by this strategy to derive irrelevant facts like *R(g,d)*, *R(f,e)* etc. We will see that communication is required to compute and store these derived facts, so avoiding irrelevant fact generation is important.

The first two properties "fall out" of the combination of bottom-up evaluation and DHT-based distributed query processing. This is a surprising and rather elegant result: the execution of a centralized query plan over a DHT produces a communication pattern similar to the one used in an explicit message-passing network protocol. We are encouraged by this example to further investigate the interplays between the deductive database literature and multi-hop routing.

Next, we consider the problem of sending irrelevant facts. There is an extensive literature on recursive query optimization to avoid sending irrelevant facts [10]. One of these techniques is *magic sets rewriting* [4] which employs program rewriting to avoid computing unneeded facts. In our case, the magic-rewritten program becomes:

R3: *magicNodes(Y) :- magicNodes(X),*
link(X,Y).
R4: *reachable(X,Y) :- magicNodes(X), link(X,Y).*
R5: *reachable(X,Y) :- magicNodes(X),*
link(X,Z), reachable(Z,Y).
R6: *magicNodes(a).*
R7: *magicNodes(b).*
Query: *?- reachable(N,M).*

The main difference after rewriting is the addition of rules for generating *magicNode* facts, which restricts the query computation to the nodes reachable from *a* and *b*.

6 Status and Preliminary Results

Currently, PIER's native query language is neither Datalog nor SQL, but a logical dataflow language for generating dataflow graphs like that in Figure 2. Recursion has been implemented in PIER, and as a proof of concept we have implemented the Gnutella crawler described in Section 4 over PIER and run it with a start set consisting of two ultrapeers. The crawl ran for 6 minutes. The link and nodes data were published back to the DHT and used to compute reachability data as described in Section 3.1. The experimental results in Figure 3 show increasing throughput (more reachable nodes discovered) as we increase the number of crawler nodes running on PlanetLab [2].

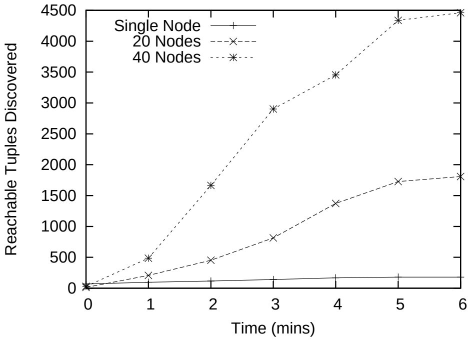

Figure 3: *Distributed Gnutella Crawler Throughput using 1, 20 and 40 PlanetLab nodes.*

7 Related Work

Query execution and optimization of recursive queries is a rich area of research. The survey paper [10] provides an excellent overview as well as references to other more advanced query processing and optimization techniques. There has also been previous work on parallel execution strategies of recursive queries [5] over static data within a parallel cluster that does not involve the multi-hop communication substrates introduced by DHTs.

Of particular relevance to our work is the use of Prolog for systems and network monitoring in InfoSpect [11] and Sophia [14]. The main distinctions between PIER and these systems is PIER's support for recursive operators, and the focus on *in-network* query processing. A secondary distinction is in the use of Prolog instead of Datalog. Prolog uses a *top-down* or backward-chaining evaluation strategy, as opposed to a *bottom-up* or forward-chaining strategy used by Datalog. There have been extensive comparisons between the two evaluation strategies [9].

8 Conclusions and Future Work

In this paper, we motivate the use of recursive queries for analyzing P2P overlays. We propose two approaches of realizing the recursive functionality. In the first approach, recursive queries are used as embedded network queries, where each node embeds query processing functionality to compute various network properties. The second approach runs distributed recursive queries over the structure of a different network. We provide a case study of executing a recursive query over DHTs, and show that the communication patterns have interesting connections with multi-hop in-network query processing, and are amenable to optimization techniques from the deductive databases literature.

Executing recursive queries in-network raises several open questions for future work. First, we suspect that the communication patterns observed in Section 5 will not always fall out nicely from traditional bottom-up evaluation strategies over DHTs. Second, DHT locality is obviously a key performance issue, and will determine whether optimized declarative queries run as efficiently as hand-coded versions. Third, the deductive database literature leaves a number of open questions in query optimization, including determining optimal join orders and base-data acquisition ("access methods"). Last, the network dynamics suggest that query optimizers should not make static decisions: network performance is hard to predict, and non-uniformities in graph topologies lead to changing join "selectivities" mid-query. Adaptive query optimization techniques like Eddies [3] may have an important role in this context.

References

[1] IrisLog: A Structured, Distributed Syslog. `http://www.intel-iris.net/irislog.html`.

[2] PlanetLab. `http://www.planet-lab.org/`.

[3] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. pages 261–272, 2000.

[4] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Sixth ACM Symposium on Principles of Database Systems*, pages 269–284, 1987.

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

[6] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sep 2003.

[7] M. Stonebraker and J.Hellerstein (Editors). Introduction to chapter 9: Vision statements. *Readings in Database Systems*, 1998.

[8] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[9] Raghu Ramakrishnan and S. Sudarshan. Bottom-Up vs Top-Down Revisited. In *Proceedings of the International Logic Programming Symposium*, 1999.

[10] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[11] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems. In *Proceedings of the 2002 ACM SIGOPS European Workshop, Saint-Emilion, France*, September 2002.

[12] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

[13] R. van Renesse and K. Birman. Scalable management and data mining using Astrolabe. In *IPTPS 2002*, 2002.

[14] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proc. HotNets-II, Cambridge, MA, USA*, November 2003.