Querying and Routing in Next-Generation Networks

Boon Thau Loo
Ph.D. Qualifying Exam Proposal
Computer Science Division
University of California at Berkeley
boonloo@cs.berkeley.edu

Abstract

I propose the use of *recursive queries* [24] as an interface for querying distributed network graph structures. Recursive queries allow a query result to be defined in terms of itself. This is particularly useful for querying network graphs that exhibit recursive structures. To query these distributed graphs over the Internet, I propose using distributed query processing techniques to process recursive queries. I further demonstrate the relationship between in-network execution of recursive queries and traditional routing protocols. Based on this relationship, I propose investigating the use of recursive queries for end-hosts to customize routing protocols. I plan to implement my proposals in the context of the PIER [9] system, and study different techniques to achieve good performance in the system.

Executive Summary
This report covers my thesis-related work to date, and presents tentative plans for the dissertation. My quals presentation will follow this document fairly closely. The report is structured as follows. After intro and background (Sections 1 and 2), I provide a brief summary of my 3 papers on p2p keyword search (Section 3), which inform my later work. Sections 4 and 5 begin the discussion of the core of the thesis, presenting a fairly detailed but preliminary report on using distributed recursive queries for routing, which is taken from my recent HotNets submission. Section 6 presents a tentative plan of action, which very likely includes more material than I will want to pursue in my dissertation.

1 Introduction

The current Internet architecture adopts *host-centric*[1] protocols defined in terms of IP addresses. The simplicity of its communication paradigm has contributed greatly to its scalability and efficiency. However, the Internet does not provide adequate support for an increasing number of applications. This includes applications that involve finding data objects whose locations cannot be easily determined within the current Internet architecture. In addition, because the routing functionality is embedded in the Internet infrastructure itself, applications have little control over the path followed by their packets. The lack of control over routing has limited the ability of the infrastructure to evolve and meet the demands of new applications or provide new services.

To address the first limitation on object location, there have been recent proposals for *data-centric* overlay networks. These overlay networks are layered on top of the existing Internet architecture. They have the important property that they achieve *data independence* [8], by allowing users to name and query data regardless of their locations. These networks also utilize decentralized peer-to-peer (p2p) protocols that allows them to run efficiently at Internet scale. Distributed Hash Tables (DHTs) [2, 27, 29, 26, 33, 25] are an example of p2p overlay networks that provide data independence at Internet scale. While DHTs solve the scalable object location problem, they do not address the second limitation on the lack of routing flexibility.

In this work, I propose exploring the synergy between *query processing* and *routing* in data-centric p2p networks. In particular, I propose the use of declarative queries, not only for *querying* networks, but also for *routing* in these networks. To demonstrate this synergy, I propose the following contributions:

- **p2p Search: A Comparison Study**. First, I ground this research by conducting a comparison study of two alternative p2p architectures, namely a more traditional flood-based approach and the use of DHTs. I based this comparison on a demanding query workload for p2p web search and file-sharing. Section 3 provides an overview of this comparison study.

- **Querying Networks with Recursive Queries.** Second, in Section 4, I propose the use of *recursive*

[1] The terms "host-centric" and "data-centric" are derived from a VLDB 2003 keynote talk given by Scott Shenker.

queries [24] as a powerful interface for querying distributed network graph structures. Recursive queries allow a query result to be defined in terms of itself. This is particularly useful for querying network graphs that exhibit recursive structures. To query these distributed graphs over the Internet, I propose the use of distributed query processing techniques to process these queries. The foundation of my proposal is the theory of recursive query processing in deductive databases [1], which focused on single-site systems. The Internet's unique execution and application environment raises many new research challenges. These include efficient in-network execution via query optimization and work-sharing techniques, and handling dynamicity in the network. I propose describe these challenges in more detail and outline some basic solutions in Section 4.

- **Customizable Routing with Recursive Queries.** Third, in Section 5, I propose the use of recursive queries for end-hosts to customize routing protocols. I demonstrate the relationship between in-network execution of recursive queries, and traditional routing protocols such as Distance Vector and Dynamic Source Routing [10]. I illustrate the flexibility of using declarative queries through several example routing protocols, and show that the query optimization and work-sharing techniques described in Section 4 are applicable here.

I intend to provide the above mentioned features in the context of the PIER [9] system. PIER is a relational query processor that is designed to run on DHTs. In Section 6, I outline the current implementation status in PIER, the research plan, and proposed timeline.

2 Background on p2p Networks

There are two basic architectures for p2p networks, namely *unstructured* and *structured networks*. Unstructured networks such as Gnutella [7] and Kazaa [11] have been widely used in file-sharing applications. These networks are organized in an ad-hoc fashion and queries are flooded in the network for a bounded number of hops (TTL).

To address the scalability issues with unstructured networks, there have been proposals to build structured networks, otherwise known as DHTs. As its name implies, a DHT provides a hash table abstraction over multiple distributed compute nodes. Each node in a DHT can store data items, and each item is indexed via a lookup key. At the heart of the DHT is an overlay routing scheme that delivers requests for a given key to the node currently responsible for the key. This is done without global knowledge or permanent assignment of the mappings of keys to machines. Routing proceeds in a multi-hop fashion; each node keeps track of a small set of neighbors, and routes messages to the neighbor that is in some sense "nearest" to the correct destination. Most DHTs guarantee that routing completes in $O(\log N)$ message hops for a network of N nodes. The DHT automatically adjusts the mapping of keys and neighbor tables when the set of nodes changes.

3 p2p Search: A Comparative Study

To better understand the performance characteristics of these two competing p2p architectures described in Section 2, we first compare their abilities to perform p2p search efficiently. While centralized search engines such as Google work well, p2p search is worth studying for the following reasons. First, search is a canonical application familiar to end-users. Second, search offers a good stress test for p2p architectures as they involve a large dataset and many concurrent queries. Third, p2p search might be more resistant then centralized search engines to censoring or manipulated rankings. Last, p2p search might be more robust than centralized search as the demise of a single server or site is unlikely to paralyze the entire search system.

While unstructured networks are effective for locating highly replicated items, they are less so for rare items. The other alternative is to use inverted indexes on DHTs. However, DHTs may incur significant bandwidth for publishing the content, and for executing more complicated search queries such as multiple-attribute queries. Despite significant research efforts to address the limitations of both unstructured and DHT search techniques, there is still no consensus on the best p2p design for searching. We perform a comparison based on two well-known p2p search workloads, namely *web search* and *file-sharing*.

3.1 Web Search

Full-text keyword search of the Web is arguably one of the most important Internet applications. It is also a hard problem, given the stringent user latency requirements (on the order of seconds) and the size of the documents involved (4 billion documents based on Google). In [14], we analyzed the feasibility of using both unstructured networks and DHTs to performing p2p web search. The feasibility studied was carried out using back-of-the-envelope calculations based on the reported query load and document sizes of Google, and the estimated bisection bandwidth of the Internet. The calculations concluded that both flooding and DHTs would consume too much bandwidth for the average web query. In fact, because of the sheer sizes of the inverted lists, it is cheaper to flood the entire network with queries, rather than use DHTs. While the DHT approach is less desirable, it is more amenable to improvements by applying well-known optimization techniques for fast inverted list intersections.

A combination of optimizations and compromises on the quality of results bring the DHT approach within feasibility range for p2p Web search.

3.2 File-Sharing

Unlike web search, file-sharing is a less demanding workload, due to the fewer number of documents and the less stringent user requirements. The inverted files are also smaller, since only filenames and metadata of files need to be indexed. Hence, given the less demanding workload, there is open debate whether DHTs are even required in this environment. We attempt to address the question on a number of fronts in the following papers [18, 20]. First, we highlight the strengths and weaknesses of unstructured p2p networks via an extensive empirical analysis of the Gnutella network. we performed live, distributed monitoring of the Gnutella network via multiple machines spread across the two continents in the PlanetLab testbed [22]. We gathered extensive traces of the network's graph structure, its query workload, and its file contents. One of the key observations is that replication of files in the network follows a long-tailed distribution with a moderate number of "popular" files containing many replicas in the network, and a long tail of many "rare" files containing few replicas. Given that observation, we observe that the flooding-based approach in unstructured networks is an efficient, simple solution for finding copies of popular files, but has poor latency and result quality for queries that focus on rare items.

Second, we describe *PIERSearch*, our implementation of DHT-based keyword querying. PIERSearch is an application built on top of PIER [9], a DHT-based Internet-scale relational query engine we have built. The DHT-based approach does provide better answers in terms of query recall, but can require more network overhead to "publish" files by keyword into the DHT, and to perform distributed joins of keyword lists at query processing time.

Based on the analysis of the workload and solutions, we propose a simple hybrid approach for high-quality p2p search, in which PIERSearch is used to build a partial index [28] over only the rare items in the Gnutella network. Queries are handled in a hybrid manner: popular items are found via the native Gnutella protocol, and rare items are found via PIERSearch.

We provide an analytical model to study the potential benefits of a universal deployment of PIERSearch bundled with Gnutella. Using this model together with the Gnutella traces, we study the trade-off between query recall and system overhead of the hybrid system. In addition, we propose and compare a variety of techniques for one of the key challenges in the hybrid solution: correctly identifying the "rare" files that should be indexed in the DHT.

Finally, we implemented this solution by modifying the open-source LimeWire [15] Gnutella software, combining it with PIERSearch. we ran the implementation on fifty PlanetLab [22] nodes across two continents, participating live in the Gnutella network; the addition of PIERSearch alongside Gnutella – even on a limited subset of Gnutella nodes – demonstrates notable benefits in both latency and recall for queries that focus on rare items.

4 Querying Network Graphs with Recursive Queries

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, hyperlink 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.

A recursive query [24] allows a query result to be defined in terms of itself. Such queries are particularly useful for querying relationships in graphs that themselves exhibit recursive structures. Declarative queries on graphs can be achieved only with recursive queries, which were a topic of intense research in database theory circles in the 80's and early '90's.

As an example application, recursive queries can be used to monitor the structural properties of Gnutella, a system that we measured extensively as described in the previous section. The type of monitoring queries include those that compute the diameter, robustness (number of paths between two nodes), and search *horizon* (nodes that are reachable within a bounded number of hops) statistics. Some useful search horizon statistics 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. The knowledge of the graph topology can also be used to improve searching in Gnutella, by routing the search query towards higher degree nodes instead of flooding.

Another example application is the use of recursive queries to monitor the structural properties of DHTs under churn. The important metrics include *dynamic resilience*, and *average path length*. *Dynamic Resilience* is the number of routable live paths between any two DHT nodes. *Average Path Length* is the average number of hops between any two DHT nodes.

Recursive queries can also be used to perform web crawls. Interested readers are referred to [21] on our prototype implementation.

4.1 Execution Model

Recursive queries can be processed either in a centralized or distributed fashion. Centralized approaches would require servers to periodically gather network information

from the infrastructure. Each query would then be sent to one or more of these servers, which would process the queries using their internal databases and return the result to the querier.

An alternative that we explore in this proposal is to execute the queries in the infrastructure in a distributed fashion. This alternative ensures that our approach scales organically with the number of nodes, and adheres to the spirit of decentralization in the Internet itself. In this case, each node runs a general-purpose query processing. An example of such a system is PIER [9].

4.2 Facts for Query Processing

To execute queries, each node maintains local information directly accessible by the local query processor. Initially, this local information consists of the properties associated with the node itself, and of the links to its neighbors. This local information is typically stored at the node itself, or available to the node via a wrapper. To keep with the terminology in deductive databases, we will refer to local information as *base facts*. Specifically, the format of the base facts is as follows:

- **node(<u>nodeID</u>, ...)**. A $node$ facts 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) [2] identifier), a web URL or a physical address (IP Address). Other fields representing node metrics (e.g. load) may also be included.

- **link(<u>source</u>, destination, ...)**. The routing table is represented as a set of *link* facts, where a $link$ fact represents an edge from *source* to *destination*. Other fields representing link metrics (e.g. delay, loss rate, bandwidth) may also be included.

Each fact is stored at the address indicated by the <u>underlined</u> address field. During query execution, the query processors generate intermediate data, called *derived facts*. Derived facts are specified by the query, and either stored locally or sent to a neighbor of the computing node for further processing. In addition to the base and derived facts, a query processor generates *result facts* that are part of the query answer.

4.3 The Basics: From Datalog to Query Plans

We begin the 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.

4.3.1 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 [24], names for facts, 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 facts 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 facts to unbound variables in the query that satisfy the rules in the program.

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

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

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

Query: *path(<u>S</u>,D,P,C).*

The above program works as follows. Rule NR1 produces new one-hop paths from existing link facts, storing them at the source node. Rule NR2 recursively produces path facts of increasing length by matching the destination fields of existing links to the source fields of previously computed paths; the new path facts are stored at at the source node. The matching is expressed using the two "Z" fields in *link(<u>S</u>,Z,C_1)* and *path(<u>Z</u>,D,P_2,C_2)* in rule NR2. Intuitively, rule NR2 expresses that if there is a link from node *S* to node *Z*, and there is a path from the same node *Z* to node *D*, then there must be a path node *S* to node *D*.

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 facts 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 bounded to constant *b*.

We now focus on the remaining gray 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 the gray additions, rules NR1 and NR2 also compute the total path costs, and the path vectors themselves.

4.3.2 Query Plan Generation

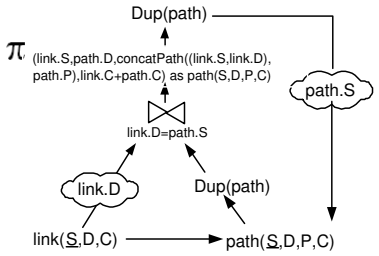

Figure 1: Query Execution Plan for the Network-Reachability Program.

Figure 1 shows a query "plan" for the Datalog program. A query plan is a dataflow diagram consisting of relational operators and arrows indicating the flow of facts. The transformation to this query plan is as follows. Rule NR1 is a simple renaming of existing link facts to path facts, 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 facts ($link.D$) with the source fields of existing path facts ($path.S$). The fields used for matching are a result of variable unification of the common "Z" fields in rule NR2. The join operator, represented by the $\bowtie$ symbol, matches link and path facts 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 facts. The *Dup* operator removes duplicate facts 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 facts from one node to another, and are labeled with the destination node. The first cloud ($link.D$) ships link facts to the nodes indicated by their destination address fields, in order to join with matching *path* facts stored by their source address fields. The second cloud ($path.S$) ships new *path* facts computed from the join back to their source nodes for further processing.

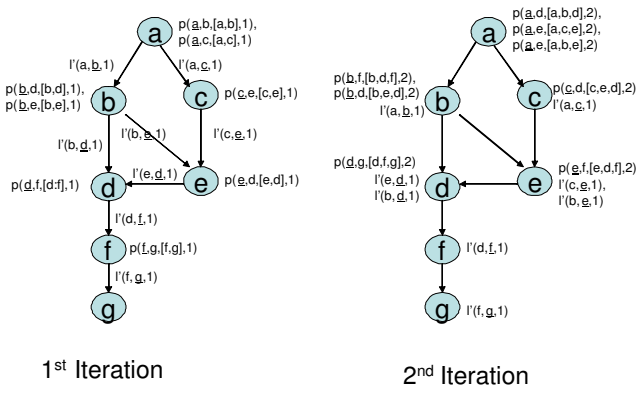

Figure 2: Nodes in the network are running the query plan in Figure 1. p(S,D,P,C) abbreviates path(S,D,P,C) *and l(S,D,C) abbreviates* link(S,D,C). *Link costs in the example are set to 1, and hence path cost is equal to the number of hops. l'(S,D,C) refers to link facts that are shipped and cached at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

4.3.3 Query Plan Execution

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

The communication occurs in stages, where each stage or *iteration* represents a "round of communication", in which all nodes exchange facts from the previous iteration. Each iteration represents the traversal of a "cloud" in Figure 1. The first iteration derives single-hop path facts from the first rule of the program. It does this by traversing the $link.D$ cloud, which ships link facts to the address in their destination field, where they are cached for the duration of the query[2]. 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 facts are joined with existing one-hop path facts to produce two-hop path facts. These facts are then sent back to the source nodes (the $path.S$ cloud) and three-hop path facts are computed. The query is completed after *k* iterations, where *k* is the diameter of the network. To illustrate further, we step through the communication necessary for the computing the path fact *p(b,f,[b,e,f],2)* for node *b*:

1. Node *b* ships *l(b,d,1)* to *d*. It is stored as *l'(b,d,1)* at node d for the duration of the query.

2. The query processor at node *d* joins *l'(b,d,1)*

[2]For undirected graphs, this iteration of shipping link facts can be avoided by adding an extra rule $link(S, D, C) = link(D, S, C)$.

and *p(d,f,[d:f],1)* to produce the new path fact *p(b,f,[b,d,f],2)*.

3. *p(b,f,[b,d,f],2)* is shipped back to node *b*.

At node *b*, *l'(a,b,1)* is joined with *p(b,f,[b,d,f],2)* to produce *p(a,f,[a,b,d,f],3)* which is shipped to node *a*.

4.3.4 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 the 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 fact maintains only the next hop on the path, rather than the entire path vector itself[3]. 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 main difference between this query and the actual distance vector computation is that rather than sending individual path facts between neighbors, the traditional distance vector method batches together a vector of costs for all neighbors.

4.4 Challenges

Unlike traditional deductive databases, network graphs are large, distributed, dynamic, and often based on soft state. These properties present new, practically grounded research challenges. The metrics that traditional deductive databases used include the number of I/Os, CPU consumption and the number of facts generated. In contrast, the metrics that are important in the distributed environment includes bandwidth communication and latency.

[3]The *W* field in DV2 represents the next-hop to node *D* from intermediate node *Z*, and can be ignored by node *S* in computing its next hop to node *D*.

Based on the *Network-Reachability* example, we have further identified the following challenges that will need to be addressed:

- **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 the work done by previous or concurrent queries be reused in order to reduce redundant work?

- **Handling Dynamicity in Networks:** Given that the network is dynamic and the queries can be long running, how can the robustness and accuracy of query results be efficiently maintained?

- **Expressing Routing Protocols:** The example above demonstrate the relationship between recursive query execution and the distance vector routing protocol. Can this language be used to express other useful routing protocols? If so, what is the expressiveness and limitations of this language?

In the remaining of this section, we will address the first two challenges, and revisit the issue of expressing routing protocols in Section 5.

4.5 Query Optimization Techniques

In this section, we address the challenge of generating efficient query plans from the declarative queries. We utilize four well-known query optimization techniques used in centralized deductive database systems, and discuss how useful they will be in generating efficient query plans in a distributed environment. They are *aggregate selections*, *magic sets rewriting*, *left-right recursion rewriting* and the *squaring algorithm*.

In addition, we address previously unexplored challenges introduced by our environment, which requires work-sharing among a diverse set of queries for scalability.

4.5.1 Pruning Unnecessary Paths

A naive execution of queries with aggregates requires the enumeration of all possible paths. This can be avoided with *aggregate selections* [30]. 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 facts 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.

4.5.2 Limited Sources and Destinations

The *Network-Reachability* example in Section 4.3 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* [3] 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*),*
P = concatPath(link(S,Z,C_1*),*P_2*),*
C = C_1 + C_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.

4.5.3 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,$\mathbf{P_1}$**,**$\mathbf{C_1}$**),**
link(Z,D,$\mathbf{C_2}$**),**
P = concatPath(P_1*, link(Z,D,*C_2*)),*
C = C_1 + C_2.
Query: *path(N,M,P,C).*

$\#include$ is a macro used to include earlier rules. 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 facts 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).

4.5.4 Reducing Result Latency

All of the previous examples generate paths of increasing hop counts at each iteration. The number of iterations required to complete the query is hence equivalent to the diameter of the network. There are alternative evaluation techniques such as the *squaring algorithm* [31], that reduces the number of iterations to logarithmic the diameter of the network, by generating paths of length 1, 2, 4, 8, etc. The reduction in iterations comes at the expense of increased messaging overhead. Squaring algorithm is generally useful for large-diameter sparse graphs.

4.5.5 Multi-Query Sharing

A key requirement for scalability is the ability to share query computation among a potentially large number of queries. We first consider sharing among queries that utilize *identical* rules. 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. To facilitate sharing among nodes issuing the same query, previously computed facts are reused whenever possible. For example, revisiting the example network in Figure 2, if node *d*'s computed path facts 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 facts 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.

Next, we consider sharing among queries with *different* rules. For example, consider running two variants of transitive closure programs with aggregates, one that computes shortest paths, and another that computes max-flow paths. We can merge these into a single variant of the Best-Path program by simply tracking two running

cost attributes (e.g., path length and path capacity) and checking two aggregate selections (e.g., min(path-length), max(capacity)).

The merged program will share all path exploration across the queries. Aggregate selections continue to be applicable, but can only prune paths that satisfy *both* aggregate selections; pruning is effective when the selections are correlated. A challenge for the query processor is to predict or discover correlated metrics across different queries, and merge rules appropriately to facilitate sharing and joint pruning.

4.6 Handling Dynamicity in Networks

In practice, query results are expected to be used for a period of time. These results 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 fact should be maintained as soft-state with an associated timeout. A smaller timeout ensures fresher results at the expense of more messaging overhead. The base facts are periodically renewed with new values, or deleted when expired. Derived facts computed using base facts should be timestamped based on the oldest base fact used in the computation, so that they expire when any of their components expire.

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

5 Customizable Routing with Recursive Queries

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 [13], enabling end-hosts to choose their paths at the AS level [32], or even computing arbitrary routes [4].

In this proposal, 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.

The use of a declarative query language has a number of attractive features for routing. First, it 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 5.2. On the other hand, it is less general than running arbitrary code in Active Networks [4], but as a result can be more safely analyzed and executed.

We also show that declarative queries need not hamper the efficiency of traditional protocols. For example, in the earlier Section 4.3, 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 in [19] 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.

5.1 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 [12]. 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 5.2 that Datalog is sufficiently expressive for a large variety of routing protocols.

5.2 Example Queries

In its purest form, Datalog has the ability to express most polynomial-time computations [12]. 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 4.3, and adding simple modifications to create new routing protocols.

5.3 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).*

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* facts 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 facts computed by rules NR1 and NR2 are stored at the source nodes, and *bestPathCost* and *bestPath* facts 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. We can further extend the rules by including constraints that enforce a QoS requirement specified by end-hosts. For example, we can restrict the set of paths to those with costs below a loss or latency threshold *k* by adding an extra constraint *C<k* to the rules NR1 and NR2.

5.3.1 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 fact 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. Other policy based decisions include not trusting the paths reported by selected nodes, or insisting that some paths have to pass through one or multiple pre-determined set of nodes.

Datalog may be suitable for expressing routing logic used in BGP inter-domain routing protocol, as proposed recently in [6]. We plan to explore this in future work.

5.3.2 Dynamic Source Routing

All of the 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 [10]:

#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).*

Rule NR1 produces new one-hop paths from existing link facts as before. Rule NR2 matches the destination fields of newly computed path facts with the source fields of link facts. This requires newly computed path facts be shipped by their destination fields to find matching links, hence ensuring that each source node will recursively follow the links along all reachable paths. The computed paths are also shipped back to the source nodes.

5.3.3 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)* facts. 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 fact. The nodes along the pathVector field for the newly produced bestPath fact are added as *avoidNodes* derived facts, and the program is executed up to k times to get k disjoint paths. To express edge-disjoint paths, instead of adding *avoidNodes*, we can add *avoidEdges* facts and modify the program to avoid these edges.

5.4 Simulation Results

We present a preliminary evaluation via simulation. The simulation computes new facts based on the query workload, and identifies those facts 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 (facts) 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.

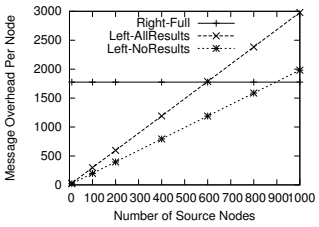

Figure 3: Message Overheads Per Node

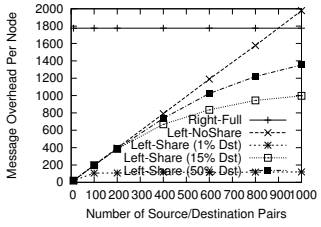

Figure 4: Message Overheads Per Node with Sharing

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 4.5.5 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. The bottom line is as follows: when there are few source and destination nodes issuing the same query, not only is the communication cost of using left recursion lower compared to computing all-pairs shortest paths, it can be further reduced with work-sharing techniques.

The 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.

6 Current Status and Research Plan

The proposed work is divided into two categories. The first category is **Infrastructure**, and this involves adding recursive query processing and optimization functionality to PIER. The second category is **Applications**, and this involves deploying applications that utilize the recursive query functionality in PIER. I aim to submit at least two papers, one for each category to conferences such as NSDI, SIGCOMM, SIGMOD and VLDB.

For the Infrastructure category, PIER currently provides the support of cycles in query plans necessary for recursive dataflows. As a starting point, I have composed "hand-optimized" PIER query plans (expressed as dataflows) for queries to compute reachability, shortest paths and network diameter queries. These "hand-optimized" dataflows have been tested and benchmarked in simulations. Detailed simulation results are available in the tech report [16]. The simulation results show that selecting the best execution strategy based on the query workload and network topology can lead to significant reduction in communication overhead and latency.

For the Applications category, I have experimented with recursive crawl queries expressed using PIER dataflows. The queries perform distributed crawls of both the Gnutella network and the web. Details of the implementations of these crawls are available at the tech reports [17, 21]. Once the network information is extracted, monitoring queries described in Section 4 on the structural properties of the networks can be issued via PIER. These queries currently run for a short period of time, and have not yet been extended to handle expiration of old data.

In view of the current status, I propose the following timeline in the next two sections.

6.1 Infrastructure Timeline

Infrastructure Phase I (Sept 2004 - Dec 2004) will gear towards adding the basic functionality in order to start developing one of the applications in its entirety. The important features in phase I include:

- **Hand-Optimized Dataflows:** This phase has been completed. We have implemented left-recursive, right-recursive, magic-sets and squaring algorithm dataflows for reachable, shortest-paths and network diameter queries. Details of the actual dataflow diagrams and simulation results are available in [16].
- **Tradeoffs in Query Plans:** I have completed a preliminary study via simulation of the tradeoffs in different query plans (represented as dataflows) while varying the number of nodes participating in the query, and the network density and size. There is more work to be done here, specifically on performing the experiments on an actual testbed like PlanetLab, and understanding the tradeoffs in greater detail.
- **Basic Sharing:** Currently, none of the work-sharing techniques discussed in Section 4.5.5 have been implemented in PIER. As a first step, I intend to implement and study work-sharing techniques among queries with identical rules.
- **Basic Support for Long Running Queries:** PIER allows queries to be executed over a period of time. However, there is remaining work left in ensuring that derived and result facts are timestamped correctly, as the network changes.

Infrastructure Phase II (Jan 2005 - Dec 2005) consist of "advanced" features that are optional, but will enhance the performance and ease of usability of applications. This phase will involve more open-ended research topics that may not be completely solved within the timeline. The features include:

- **Datalog parser:** This is a straightforward implementation of a Datalog parser. I intend to take a ready-made parser used in traditional deductive database systems [23, 5] and make modifications accordingly.
- **Automatic Plan Generation:** Given a Datalog program, I intend to explore techniques to generate an efficient query plan automatically. I expect that my investigation will raise several interesting issues as there are several variables affecting the performance of any given plan. My goal is to conduct a reasonably good study of the tradeoffs of different plans as the query workload and network conditions change, and provide sufficient mechanisms towards generating "good" plans. I expect that there will be room for future work in this area beyond the dissertation.
- **Enhanced Sharing:** This involves sharing across different queries with either some common datalog rules or rules that involve correlated metrics. I consider this an advanced feature that is only studied in greater detail if time permits.

6.2 Applications Timeline

Applications Phase I (Nov 2004 - Aug 2005) consist of the following:

- **Customizable Routing Infrastructure:** This is the main driving application that exercises most of the query optimization and work-sharing techniques described in Section 4. Besides infrastructure support, there remains much work to be done here, including a DHT wrapper to extract routing tables of each node, tools to measure link metrics, etc. I hope to leverage current tools available in building a Routing Service for the Internet Indirection Infrastructure (i3) [13] project. I expect this application to take up the bulk of my time in the applications phase.
- **Gnutella Monitoring Tool:** With much of the distributed crawler infrastructure in place, there are two main features required for a long-running version of the crawler. First, the crawler now performs a naive distribution by IP address to parallelize the crawl among PIER nodes. I intend to explore the use of a partitioning scheme based on geographic proximity. Second, the distributed crawler should not flood the Gnutella network with messages. One of the challenges with PlanetLab is to use its power carefully: early on, the experiments on performing a distributed crawl of Gnutella raised warning flags among system administrators because they resembled malicious network behavior. Appropriate rate-limiting features need to be in place for the crawler to work over a long period of time. I hope that this monitoring tool can generate interest among the Internet measurement community.

Applications Phase II (Sept 2005 - Dec 2005) consists of optional applications. These include the decentralized focused web crawler, and a comprehensive study the performance of DHTs under churn using long-running recursive queries. Once the infrastructure is stable, I intend to

propose these applications as advanced undergraduate or graduate level class projects to be worked on in collaboration with other students.

A key part of the success of this thesis is to enable my ideas to influence the networking community. As a start, I have submitted a HotNets paper advocating the use of recursive queries for routing. We also hope to have at least one of the applications running as a service on PlanetLab.

6.3 Wrap up

I have allocated the remaining six to eight months, starting from Jan 2006 to wrap up my thesis work, write the dissertation and attend job interviews.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, Feb. 2003.

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

[4] 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.

[5] M. Derr, S. Morishita, and G. Phipps. Design and Implementation of the Glue-NAIL Database System. In *Proceedings of ACM SIGMOD*, 1993.

[6] N. Feamster and H. Balakrishnan. Towards a Logic for Wide-Area Internet Routing. In *Proceedings of FDNA-03*, 2003.

[7] Gnutella. `http://gnutella.wego.com`.

[8] J. M. Hellerstein. Toward Network Data Independence. In *SIGMOD Record 32(3)*, 2003.

[9] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of VLDB*, Sep 2003.

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

[11] Kazaa. `http://www.kazaa.com`.

[12] P. Kolaitis and M. Vardi. On the Expressive Power of Datalog: Tools and a Case Study. In *Proceedings of Symposium on Principles of Database Systems*, 1990.

[13] K. Lakshminarayanan, I. Stoica, and S. Shenker. Routing as a Service. Technical Report UCB-CS-04-1327, UC Berkeley, 2004.

[14] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS 2003*.

[15] Limewire.org. `http://www.limewire.org/`.

[16] B. T. Loo. Querying Network Graphs with Recursive Queries. Technical Report UCB-CS-04-1332, UC Berkeley, 2004.

[17] B. T. Loo, J. M. Hellerstein, R. Huebsch, T. Roscoe, and I. Stoica. Analyzing P2P Overlays with Recursive Queries. Technical Report UCB-CS-04-1301, UC Berkeley, 2004.

[18] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet Scale Query Processor. In *Proceedings of VLDB*, Sep 2004.

[19] B. T. Loo, J. M. Hellerstein, and I. Stoica. Customizable Routing with Declarative Queries. Submitted to Hotnets-III.

[20] B. T. Loo, R. Huebsch, I. Stoica, and J. Hellerstein. The Case for a Hyrid P2P Search Infrastructure. In *IPTPS 2004*.

[21] B. T. Loo, S. Krishnamurthy, and O. Cooper. Distributed Web Crawling over DHTs. Technical Report UCB-CS-04-1305, UC Berkeley, 2004.

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

[23] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL Deductive System. In *The VLDB Journal, 3:161–210*, 1994.

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

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[26] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *UC Berkeley Technical Report UCB//CSD-03-1299*, Dec 2003. Revised version to appear in 2004 USENIX Annual Technical Conference, June-July, 2004.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[28] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, pages 420–427, 1995.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

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

[31] P. Valduriez and H. Boral. Evaluation of Recursive Queries Using Join Indices. In *Proceedings of the 1st International Conference on Expert Database Systems*, 1986.

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

[33] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.