Querying Network Graphs with Recursive Queries

Boon Thau Loo
boonloo@cs.berkeley.edu
University of California, Berkeley

Abstract

This paper describes a distributed infrastructure for querying network graphs with recursive queries. We argue that recursive queries have great practical value as a declarative interface to multi-hop networks. To query these networks in a distributed fashion, we describe the processing of recursive queries using PIER, a P2P relational query processor that utilizes distributed hash tables (DHTs). We focus on studying a set of commonly-used recursive queries based on the transitive closure query. We demonstrate that different query processing techniques will lead to tradeoffs in latency, work sharing and communication overheads. Our experimental results also show that selecting the best execution strategy based on the query workload and graph topology can lead to significant reduction in communication overhead and latency.

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, 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 [12] 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 1980's and early '90's.

Using example network applications, we argue 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. In this environment, the main bottlenecks are due to network communications. These properties present new, practically grounded research challenges that we intend to explore.

In all of our example applications, the queried graphs are large, distributed and dynamic. These applications are also decentralized, and it is natural to query them in a decentralized fashion *in the network*. To perform decentralized processing, we will make use of PIER [8], a relational query engine that uses Distributed Hash Tables (DHTs) [2]. Figure 1 provides an overview of the architecture. The recursive queries are expressed by applications using *Datalog* [12] programs, which are parsed, rewritten and expressed as a query plan consisting of a series of relational joins by the *Recursive Query Engine*. After query optimization, the query plan is sent to PIER for execution. The *Publisher* component is responsible for publishing the graph into PIER for querying.

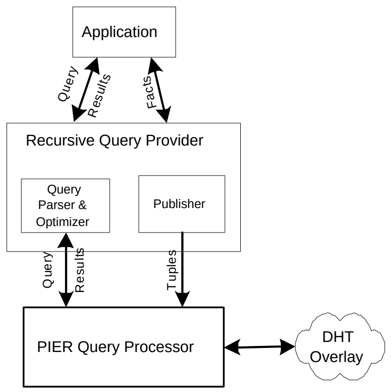


Figure 1: *Recursive Query Engine Architecture.*

To better understand the challenges that this new environment presents, we study the execution strategies for a set of commonly-used recursive queries using PIER. These queries are variants of the *transitive closure* query, which can be used to compute reachability information within a graph. We will further show how our execution techniques can be generalized to support dynamic graphs, and to facilitate work sharing across queries.

Through extensive measurements obtained by running recursive queries over PIER running on top of a network simulator, we demonstrate that picking the best execution strategy based on the query workload and graph topology can lead to significant improvements in communication overhead and latency. Given that both query workload and the network are dynamic, this requires the use of query optimizations techniques not previously explored in centralized processing of recursive queries.

The organization of this paper is as follows. In Section 2, we provide a background on DHTs and Datalog. Using Datalog programs, we highlight the usefulness of recursive queries through several applications described in Section 3. In Section 4, we describe how recursive queries can be used to query static graphs using PIER, and explore a variety of execution strategies and query optimization techniques. We then extend the discussion for dynamic graphs in Section 5. We present simulation experimental results in Section 6. Lastly, we discuss future work in Section 7 and conclude in Section 8.

2 Background

There have been many proposals for DHT designs in the last few years; we briefly describe their salient features here. An overview of DHT research appears in [2].

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 $O(\log N)$ P2P 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.

The DHT forms the basis for communication in PIER. With the exception of query answers, all messages are sent via the DHT routing layer. PIER also stores all temporary tuples generated during query processing in the DHT. The DHT provides PIER with a scalable, robust messaging substrate even when the set of nodes is dynamic.

2.1 Datalog

We give our examples using *Datalog* [12] programs, where each program consists of a set of rules and queries. We focus on programs with recursion, although Datalog can of course be used to express non-recursive 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 [12], names for predicates, function symbols and constants begin with a lowercase 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[1]. 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.

In summary, the schema of the following facts which we will be using throughput the paper is as follows (primary keys are *underlined*):

- `node(`<u>`nodeID`</u>`, ...)`. A *node* tuple stores information on a node in the network. The *nodeID* field is typically the IP address or DHT identifier of the node. It also serves as the publishing (index) key for the DHT. Other fields on node-specific information such as the hop count from source nodes, bandwidth, load, etc may also be included.
- `link(`<u>`source, destination`</u>`, ...)`. A *link* tuples represents an edge from *source* to *destination*, where both of these fields are nodeIDs of nodes in the network. Other fields such as link cost may also be included. Typically, the publishing key is the *source* field, which is true for most routing applications.
- `reachable(`<u>`source, destination`</u>`, ...)`. A *reachable* tuple represents that *destination* can be reached from *source*. Like *links*, these are typically keyed at the source node.
- `path(`<u>`source,destination, path`</u>`, cost)`. A *path* tuple represents that *destination* can be reached from *source* along the path indicated by *path*, where *cost* is the sum of all *link* costs along the path.

Tuples are constructed for these facts. They are then published and stored in the DHT based on their publishing key. By using the source field as the publishing key, we assume that the query processor at node *X* has access to *X*'s routing table.

The textbook example of recursive query is graph transitive closure, which can be used to compute network reachability. The following simple *Reachable* program computes the set of all nodes reachable from node *a*. In all our examples, *X*, *Y*, *C* and *P* abbreviates the *source*, *destination*, *cost* and *path* fields respectively.

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 node *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.

In our second example, 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 [16]. We will revisit this optimization in Section 4.2. 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 Application Scenarios

In this section, we will illustrate the use of recursive queries as a powerful tool for understanding and controlling the structural properties of network graphs. They also form generic substrate for developing more flexible and powerful routing protocols.

We consider two possible settings for realizing recursive query functionality: *external network queries* and *embedded network queries*.

In the case of external network queries, we assume that the query is executed on a *separate* DHT-based query infrastructure

[1]In this paper, we will use the terms "facts" and "tuples" interchangeably.

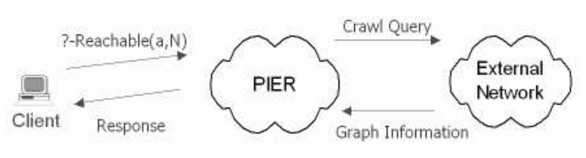
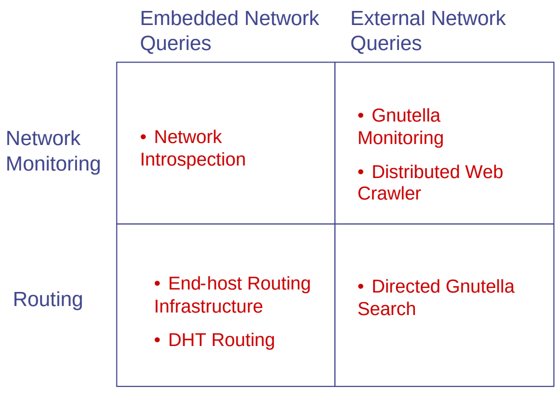


Figure 2: *External Network Queries. PIER is used to query the external network.*

such as PIER. This is illustrated in Figure 2, where PIER is used to query the external network. Since these two networks are separated, PIER has to first extract graph information from this external network via a *crawl query*. The graph information consisting of *node* and *link* facts are published into the DHT based on their publishing keys. Clients than query the external graph by sending the recursive query to PIER which processes the query on behalf of the external network.

In contrast, with embedded network queries, PIER and the external network are the same network, as each node in the external network also runs a PIER node. With embedded network queries, we assume that each node in the network embeds query functionality, and has access to the local node's routing tables. In some instances of the introspection-based applications, even the queries may be self-generated not by a third-party client but by the network itself.



	Embedded Network Queries	External Network Queries
Network Monitoring	• Network Introspection	• Gnutella Monitoring • Distributed Web Crawler
Routing	• End-host Routing Infrastructure • DHT Routing	• Directed Gnutella Search

Figure 3: *Classification of Applications.*

On top of classifying queries, there are also two classes of applications, *measurement* and *routing*. Figure 3 summarizes all the applications along these two dimensions. We will describe each one of these applications below.

3.1 Gnutella Monitoring

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.

R8: *node(X,0) :- startset(X).*
R9: *link(X,Y) :- node(X,Hop), gnutellaPing(X,Y).*
R10: *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.

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.

3.1.1 Distributing the Crawl

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.

3.1.2 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 2.1. For example, based on all-pairs-shortest-path computation, we can compute the diameter of the Gnutella network.

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. 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.

3.2 Distributed Web Crawler

The Gnutella crawl query can be modified to express a distributed crawler for the web. In this massively distributed web crawler, end-hosts donate their excess bandwidth and computation resources to crawl the web. The distributed crawler works by expressing crawl as a recursive query, which is executed by PIER. Expressing crawl as a query allow for *focused crawlers* [14], where users can specify via user-defined operators which web pages to begin crawling from, the order in which pages are crawled and which pages to avoid crawling. A decentralized crawler shared by many users also provides the possibility of collaborative filtering to aid the focused crawler. We currently have a prototype implementation of the Distributed Web Crawler. Interested users are referred to the technical report [5] for more details.

3.3 Network Introspection

Recursive queries can be used by networks to perform self-introspection. The types of queries are *embedded network queries*, and the queries are self-generated by the network to monitor its own performance or correctness.

We will provide an example based on DHTs. An important measure a DHT is its ability to handle network 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.

Current approaches for evaluating DHTs include the use of simulations [6] or benchmarking the DHT as a blackbox using higher-level applications [13]. Both of these methods have limitations. By exposing the routing information of each DHT to its local PIER node, we can query the structural properties of a DHT, given its routing algorithm This allows us to accurately measure the structure of a DHT under network churn.

3.4 Routing Infrastructure

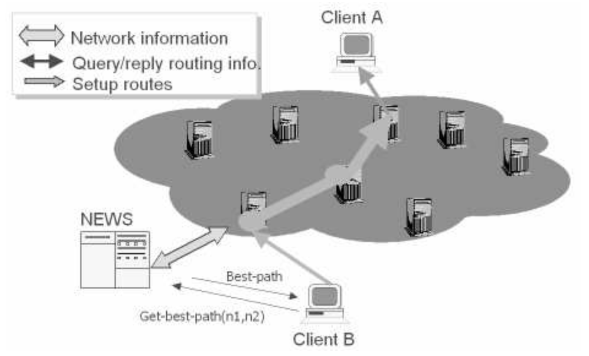


Figure 4: *Routing Infrastructure for i3*

Applications demand greater flexibility in route selection. For example, depending on the quality of service required by an application, an application may pick a best path based on latency or bandwidth. Another application may be willing to accept any paths that falls below a certain threshold latency.

A shared infrastructure for routing has been proposed in the Internet Indirection Infrastructure [10] project. In this infrastructure, network weather service nodes (NEWS) are used to gather information on the network topology. As shown in Figure 4[2], when client A wishing to route to client B, it queries the NEWS server which returns the best path to client B.

By expressing a crawl as a query, and performing the path formation *in network*, we can avoid having to centralize the graph information at the NEWS server, which leads to better scalability and avoids a single point of failure.

4 Querying Static Graphs

In this section, we examine different recursive query execution strategy over static graphs. We will revisit long running queries over dynamic graphs in Section 5.

When a query is issued and executed using PIER, each PIER node that receives the query will execute the query based on the existing *link* and *node* facts reflecting the state of the network at the time the query is received at each node. Query execution may require the movement of data in the network, and in order to take into account network delays, each query has a query duration after which the query timeouts and is removed from the system. We assume that the *queried graph* is static over the duration of the query.

All queries are executed using Datalog's bottom-up [11] ("forward-chaining") mechanism, which starts by applying the rules in the program to all existing *link* facts. In practice, we expect that our queried graphs can be either directed and undirected. While networks such as Gnutella are undirected, there are also instances of directed graphs such as the Web and some DHTs. Even when networks are undirected, the routing protocol need not be. For example, queries are only routed from regular nodes to ultrapeers, but not vice versa.

Figure 5 summarizes the various query execution strategies and optimizations that will be explored in this section. The query is expressed as a Datalog program consisting of a set of rules (Section 2.1). It is first parsed and rewritten in the following ways. First, depending on the graph topology and query workload, the query may be rewritten as *left* or *right recursive* (Section 4.1.2) to take advantage of work sharing. Second, *magic sets rewriting* (Section 4.1.1) is used to avoid computing redundant facts. Third, to prevent cycles, extra rules may be added to the query(Section 4.1.4).

Following the rewriting phase, a query execution plan consisting of relational operators is generated. Once the evaluation option (*semi-naive* or *smart*) is determined (Section 4.1.3), converting the Datalog program to query execution plan involves making decision on which fields to index the intermediate facts on, what the join ordering in the query plan and where to place duplicate elimination operators to reduce communication and computation overhead.

There are more advanced optimizations available to further reduce redundant work, by combining common sub-expressions and rehashes (Section 4.1.1). Rehashes can also be reordered to avoid hotspots and bottlenecks during query computation (Sec-

[2]Courtesy of Karthik Lakshminarayanan's Sahara retreat presentation.

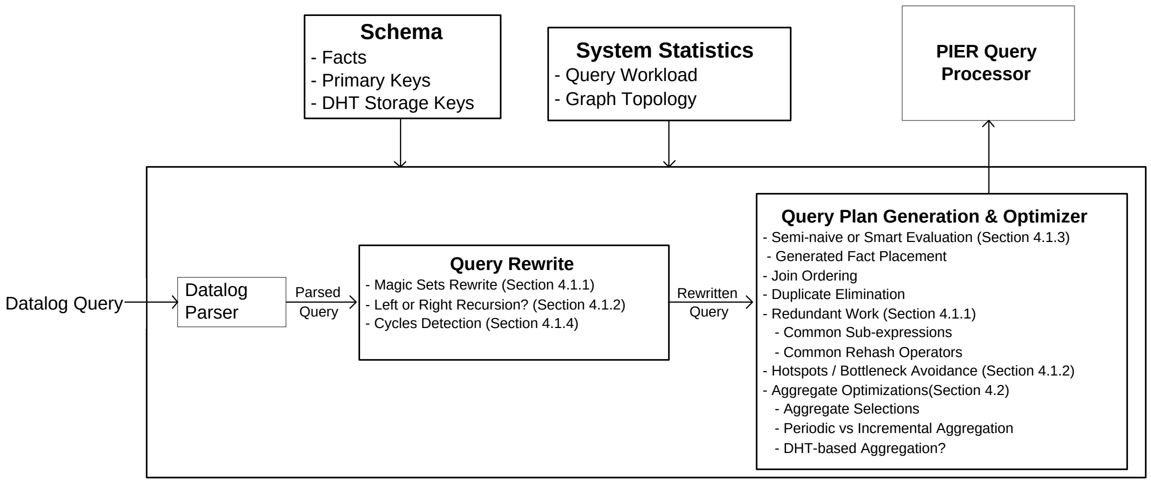

Schema
- Facts
- Primary Keys
- DHT Storage Keys
System Statistics
- Query Workload
- Graph Topology
PIER Query Processor
Datalog Query
Datalog Parser
Parsed Query
Query Rewrite
- Magic Sets Rewrite (Section 4.1.1)
- Left or Right Recursion? (Section 4.1.2)
- Cycles Detection (Section 4.1.4)
Rewritten Query
Query Plan Generation & Optimizer
- Semi-naive or Smart Evaluation (Section 4.1.3)
- Generated Fact Placement
- Join Ordering
- Duplicate Elimination
- Redundant Work (Section 4.1.1)
- Common Sub-expressions
- Common Rehash Operators
- Hotspots / Bottleneck Avoidance (Section 4.1.2)
- Aggregate Optimizations(Section 4.2)
- Aggregate Selections
- Periodic vs Incremental Aggregation
- DHT-based Aggregation?

Figure 5: *Datalog Query to Optimized Query Plan.*

tion 4.1.2). We will also discuss a number of optimizations available to recursive queries dealing with aggregates in Section 4.2.

We will evaluate different execution strategies and optimizations using the following metrics:

- **Number of Iterations:** We define an iteration as a "round of communication", where existing facts are *rehashed* to compute new facts. Rehashing involves republishing existing facts back into the DHT based on a field value different from its original storage key. As we shall demonstrate later, rehashes are necessary in order to compute new facts. The new facts that are produced are then rehashed in the next iteration to generate more facts. The number of iterations will affect the *fixpoint latency*, which is defined as the time taken for all result facts to be inferred from existing rules. This is also known in deductive database literature as a *fixpoint*. In practice, the fixpoint latency will also be affected by networks delays and the topology of the queried graph.

- **Communication Overhead:** An efficient query plan minimizes communication overhead. We base this metric on the number of messages rehashed during query execution. These messages include intermediate data which needs to be sent to other nodes for further computation, or result facts which needs to be stored at appropriate nodes in the network.

- **Intra-Query Work Sharing:** An effective execution strategy prevents redundant work from being carried out on the same portion of the graph. Typically, this redundant work arises from overlapping paths, and co-ordination is non-trivial since the nodes performing the computation are distributed. We term this ability to prevent redundant work within a query *intra-query work sharing*. Note that this differs from *inter-query work sharing* which shares query results *across* different queries. We will discuss the latter in Section 5.

4.1 Transitive Closure Query

We revisit the transitive closure query: the *Reachable* query introduced in Section 2.1 that computes the set of *reachable* nodes. There are two types of transitive closure computation that are of interest to us. The first is *full transitive closure*, where all-pairs reachability information is computed on the entire graph. The second is *partial transitive closure*, where only a portion of the graph is queried. Typically, restrictions are placed on either the source or destination nodes. Without loss of generality, we will consider limitations by *source nodes* only, but the same conclusions applies to limits on destination nodes.

The following *Reachable* query does not impose a restriction on either source or destination, and hence computes the *full transitive closure*:

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

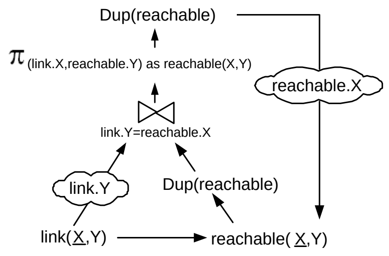


Figure 6: *Query Execution Plan for the Reachable Program.*

The recursion is *right-recursive*, since the recurring term appears as the second term (hence the "right") in the body of R2[3]. Figure 6 shows the right-recursive query plan for computing full reachability beyond the first hop. The DHT storage key is underlined in the query plan. The clouds in the figure represent rehashing (publishing) tuples into the DHT and are labeled

[3]We will revisit the *left-recursive* version in Section 4.1.2.

with the keys used to rehash them. The *Dup* operator removes duplicates from the input tuples to the operator.

The translation from Datalog to the query plan is as follows: R1 is simply a rename of existing *link* facts as *reachable* facts. Based on rule R2, the query plan outputs a new *reachable* fact whenever there is a *link* fact whose *destination* matches a *source* of an existing *reachable* fact. This is expressed as a join in Figure 6. New *reachable* facts that are computed from the join are then rehashed by their source fields for further processing. Duplicate operators are added to ensure that duplicate facts are not rehashed or recomputed twice.

Using this query as our example to compute the *full transitive closure*, we 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. The communication patterns for computing the full transitive of a graph is illustrated in Figure 7. *r(X,Y)* abbreviates *reachable(X,Y)* and *l(X,Y)* abbreviates *link(X,Y)*.

In the 0^{th} iteration (not shown in the figure), *reachable* facts of a single hop are derived from R1 of the Reachable query. For example, *r(a,b)* is derived from *l(a,b)*. In the 1^{st} iteration, *link* facts are rehashed by their *destination* fields. In the 2^{nd} iteration, the rehashed *link* facts are joined with existing *reachable* facts to produce two-hop *reachable* facts. These facts are rehashed back to the source nodes. In the 3^{rd} iteration, new three-hop *reachable* facts are produced from the two-hop *reachable* facts and rehashed by their source nodes to generate more *reachable* facts. In practice, querying query execution, the various iterations may overlap. E.g., *reachable* facts may be produced from the 2^{nd} iteration as soon as some *reachable* facts are produced from the 1^{st} iteration.

The computation of this 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 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(a,h)* for node *a*.

1. *a* rehashes *l(a,c)* to *c*.
2. *c* rehashes *l(c,e)* to *e*.
3. *e* joins *l(c,e)* and *r(e,f)* and rehashes result *r(c,f)* to *c*.
4. *c* joins *l(a,c)* and *r(c,f)* and rehashes result *r(a,f)* to *a*.

We evaluate based on the three metrics:

- **Number of Iterations:** Each increasing iteration produces *reachable* facts who source and destination nodes are one hop further apart in distance. No new *reachable* facts are produced after the 5^{th} iteration, which is equals to the length of the longest path in the network. For undirected graphs, we can avoid the rehash of the initial *link* facts and this reduces the number of iterations by one.
- **Intra-Query Work Sharing:** Many nodes in our example share the common reachable node *f*, and the path $f \rightarrow h \rightarrow i$. This query plan ensures that we only need to compute *reachable* facts sourced at node *f* once. To illustrate, both *reachable(f,h)* and *reachable(f,i)* facts are computed once and stored at node *f*, and subsequently used by nodes *a*, *b* and *c* to compute their reachability facts. Such storage and sharing of intermediate results is also known in deductive databases as work *memoization*. This sharing happens because the *reachable* facts sourced at node *f* is used by all nodes that can reach node *f* to compute their *reachable* facts. The use of duplicate elimination at node *f* ensure that only one *reachable(f,h)* fact is used during the computation of new *reachable* facts. This work sharing is achieved as a result of using right-recursive execution techniques with duplicate elimination. We shall show later that such work sharing does not occur for when the *left-recursion* execution strategy is used.
- **Communication Overhead:** The cost of computing the one-hop *reachable* facts is zero, since it can be directly inferred locally from each node's links. Each additional reachable fact require two messages (as a direct benefit from work sharing), one to rehash *link* facts by its *destination* field, and the other to send the computed *reachable* facts by the *source* field for further processing. In our example above, there are 19 *reachable* facts that are two hops or greater. Given that there are 11 distinct links, the total number of messages required to compute the full-transitive closure is $19 + 11 =$**30** messages.

The above 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.

4.1.1 Magic Sets

The execution strategy in Section 4.1 computes the full transitive closure. Suppose instead, we are only interested in querying a portion of the graph limited to nodes reachable from *b* and *e*. The naive solution would be to compute the full transitive closure and then disregard *reachable* facts not sourced at either *b* or *e*. This would require rehashing links such as *link(a,b)* and *link(c,e)* that are not required by the query. If the number of reachable nodes from *b* and *e* is small relative to the size of the graph, the redundant communication overhead can be considerable.

There is an extensive literature on recursive query optimization to avoid sending irrelevant facts [12]. One of these techniques is *magic sets rewriting* [4] which employs program rewriting to avoid computing unneeded facts. In our case, if the source nodes are *b* and *e*, 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(b).*
R7: *magicNodes(e).*
Query: *?- reachable(N,M).*

We call this program the *Magic Reachable* program. The rewritten program limits by source nodes. The main difference after rewriting is the addition of rules for generating *magicNode* facts, which restricts the query computation to the nodes

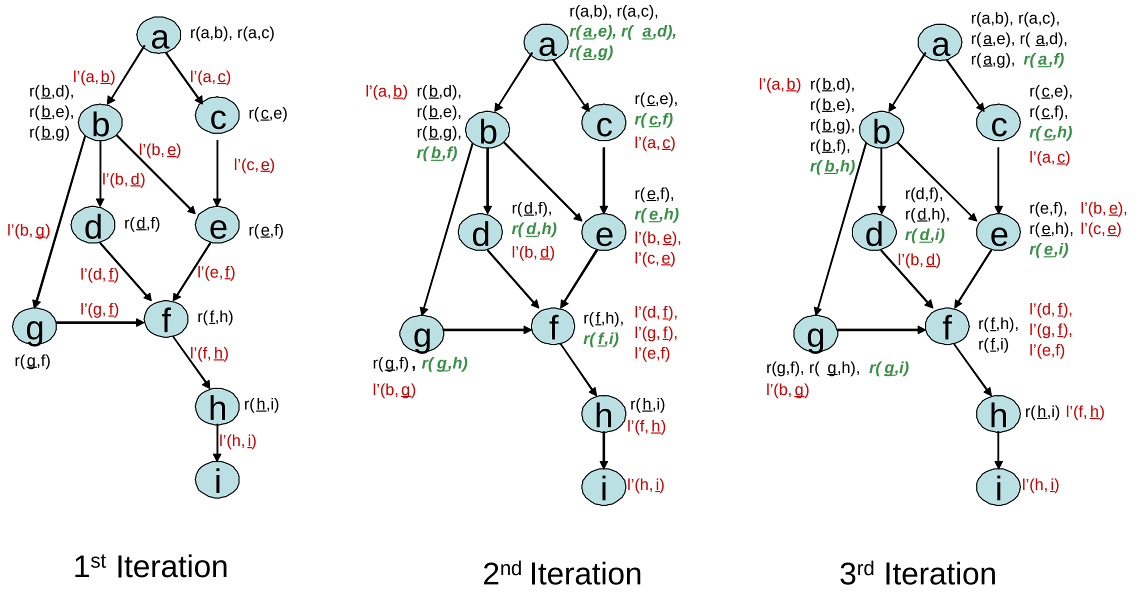


Figure 7: *The communication patterns for executing the full transitive closure using the query plan in Figure 6. In the* 1^{st} *iteration,* link *facts (shown as l'(X,Y)) are rehashed in the direction of the edges towards the destination nodes. In the* 2^nd *iterations and beyond, these rehashed links are cached on the destination nodes for the duration of the query. Due to space constraints, we omitted* link *facts keyed at the source nodes. New* reachable *facts generated at each iteration are in* italics and bold.

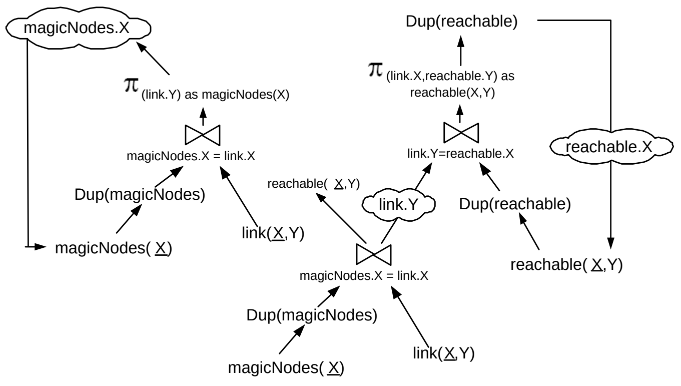


Figure 8: *Query Execution Plan for the Magic Reachable Program.*

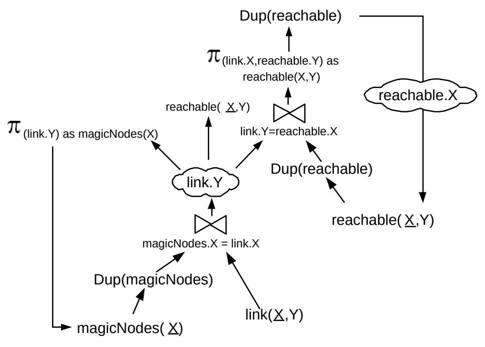


Figure 9: *Optimized Query Execution Plan for the Magic Reachable Program.*

reachable from *b* and *e*[4].

The resulting query plan of this *Magic Reachable* program is showed in Figure 8. The join ordering follows the order in which the terms in the rules appear from left to right, i.e. performing the join between *magicNodes* and *link*, followed by a join with *reachable*. This turns out to be the optimal ordering, although the query optimizer may overwrite the ordering provided by the Datalog rules. The above plan can be further optimized by eliminating redundant work:

- **Common Sub-expressions:** We observe that rules R4 and R5 share the common expression *magicNodes(X)*, *link(X,Y)* is used to compute both *reachable(X,Y)* and also *magicNodes(X)*. We will factor out this expression and only perform the join computation once in the query plan.
- **Common Rehash Operators:** we observe that within the same query plan, *link(X,Y)* is rehashed twice, first as a direct *rehash(link(X,Y))* for R5, and the second as a projection $\pi_{link.Y}$ followed by a *rehash(magicNodes.X)* for R3. We can *combine* these two rehashes to reduce communication overhead. The resulting plan rehashes *link(X,Y)* to node *Y* where *magicNodes(Y)* is computed locally (for R3), and *link(X,Y)* is used in a join with other *reachable* facts (for R5). Note that this optimization requires that one of the rehash operators be "pushed down" before the projec-

[4]The query can also be written to limit destination nodes, but this would require expressing the initial reachable program with left-recursion described in Section 4.1.2.

tion $\pi_{link.Y}$ operator in the query plan. While performing the projection before the rehash would also reduce communication, it prevents the rehash operators from being merged.

Figure 10 shows the communication patterns of the query plan in Figure 9. There are two phases that occurs in step-wise fashion where facts produced from the 1^{st} phase is immediately used for the 2^{nd} phase.

The 1^{st} phase is know as the *magic node discovery phase*. In this phase, the initial facts *magicNodes(b)* and *magicNodes(e)* are first published. At each iteration, new *magicNodes* of increasing hop count from the source nodes are discovered. When node *X* receives a rehashed *magicNode(X)*, it identifies itself as one the *magic nodes* that can be reached from any of the source nodes .It then rehashes all its outgoing links by their destination. The rehashed links are used to identify new magic nodes within this phase, and also used in the next phase. The second *reachability computation phase* involves computing *reachable* facts from links rehashed by magic nodes. The computation is similar to that described in Section 4.1. Hence magic sets enables only relevant links belonging to the magic nodes to participate in computing *reachable* facts.

Using the three metrics, we make the following observations:

- **Number of Iterations:** In general, if only a small portion of the graph is involved in the query, the use of magic sets will ensure far fewer iterations by focusing on only the relevant portion of the graph. On the other hand, if a large portion or the entire graph is involved in the query, the use of magic sets would require more iterations than computing full-transitive closure. This is because the *reachable* facts for a node can only be computed when a node is considered a magic node. The 1^{st} iteration consists of publishing the initial *magicNode* facts into the DHT. The rehash of links for a node only happens when a node receives a *magicNode* fact with the *nodeID* field set to its identifier. For example, the rehash of *link(f,h)* does not occur until the 3^{nd} iteration when node *f* receives *magicNode(f)*. In contrast, without the use of magic sets, *link(f,h)* is rehashed the moment the query is received in the 1^{st} iteration. As a result, *reachable(b,h)* is computed only few iterations later, i.e., in the 5^{th} iteration instead of the 3^{rd}.
- **Intra-Query Work Sharing:** Using magic sets is orthogonal to work-sharing. Hence, this query achieves work-sharing of common paths observed in Section 4.1.
- **Communication Overhead:** Magic Sets allow a query to be limited only to the relevant portion of the graph. In this example, the main savings are resulted from avoiding the rehash of *link* and *reachable* facts sourced at non-magic nodes *a* and *c*. This is a total savings of 12 messages, hence reducing the number of messages to 18. However, magic sets incurs the additional cost of publishing the initial magic node facts *magicNodes(b)* and *magicNodes(e)*, hence increasing the total number of messages to **20**. While the overall savings in this example is not significant, our experimental results in Section 6 shows that the savings can be immense when only a small part of the graph is queried. In general, it pays off to utilize magic sets, because even in the extreme case when the entire graph is queried, the communication overhead is no greater than the cost of computing the full transitive closure, and publishing the initial magic node facts.

4.1.2 Left Recursion

We make the observation that even when we rewrite the right-recursive query plan using magic sets to limit the query to source nodes *b* and *e*, the query computes *reachable* facts for nodes (*d*, *f*, *g*, *h* and *i*) as a side-effect. An alternative query plan that utilizes *left-recursion* avoids this extra computation. We express the *Left Reachable* program as follows:

R8: *reachable(X,Y) :- link(X,Y)*
R9: *reachable(X,Y) :- reachable(X,Z), link(Z,Y)*
Query: *?- reachable(b,N).*
Query: *?- reachable(e,N).*

In the above plan, the magic sets rewrite is done trivially by limiting all *source* field values to the set of source nodes required by the query.

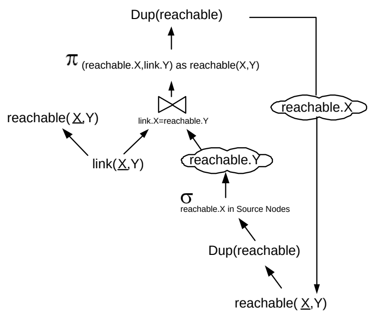


Figure 11: *Query Execution Plan for the Left Reachable Program.*

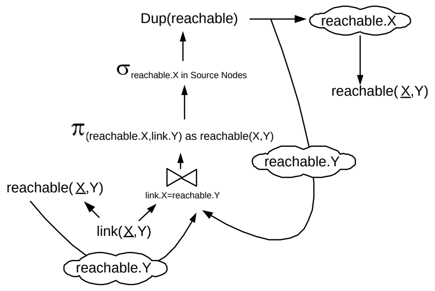


Figure 12: *Alternative Query Execution Plan for the Left Reachable Program. At each iteration,* reachable *facts are rehashed by their destination fields (shown as r'(X,Y)), joined with existing* link *facts and further rehashed.*

Figure 11 shows the query execution plan for the *Left Reachable* program. This query plan requires all computed *reachable* facts to be rehashed twice in order to produce new *reachable* facts via the join operator. This has the following drawbacks. Two rehashes (by *reachable.X* and *reachable.Y* are required to produce new *reachable* facts. This is inefficient, and also poses a possible hotspot and computation bottleneck. Figure 12 shows an alternative plan that branches the two rehashes, *rehash(reachable.X)* for indexing the *reachable* facts to be keyed

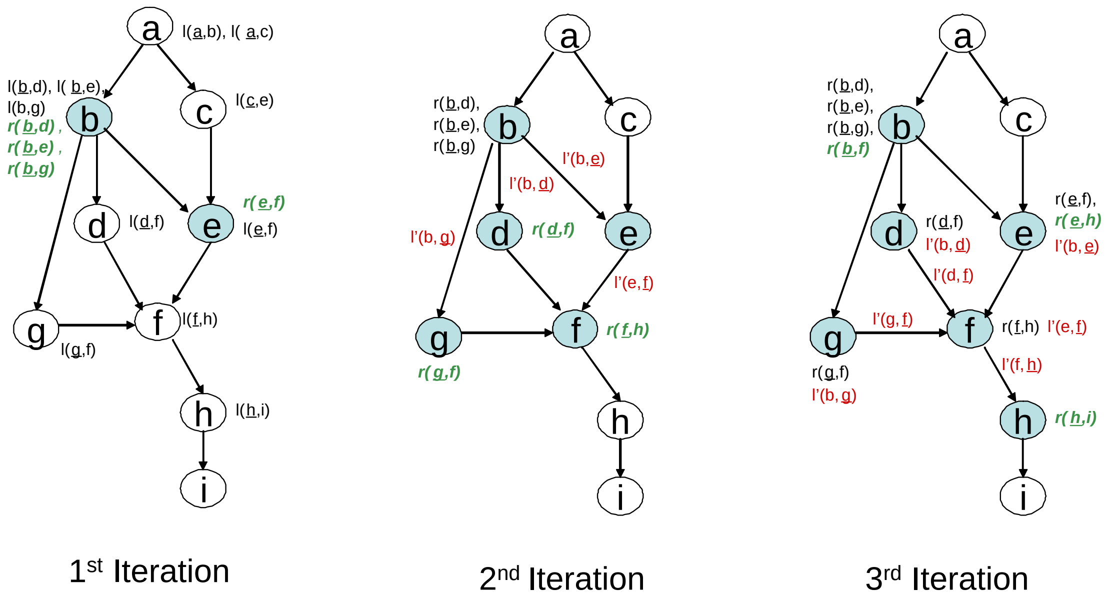


Figure 10: *The communication patterns as a result of executing the query plan for the Magic Reachable program shown in Figure 9. The shaded nodes are those identified as* magicNodes *at each iteration.*

at the source nodes, and *rehash(reachable.Y)* for computing more *reachable* facts.

Figure 13 shows the corresponding communication patterns as a result of executing the alternative query plan. The query converges in the same number of iterations as the right recursion version of the plan. For each source node, each reachable node incurs overhead for rehashing *reachable* facts by destination for further processing, and rehashing *reachable* facts by source for indexing. The former is dependent on the number of distinct reachable outgoing links, and the latter by the number of distinct reachable nodes that are two hops or greater away from the source nodes. Hence, **11** messages are required to compute the *reachable* facts for node *b*.

Unlike the right recursive version of the query, the left-recursive version does not achieve intra-query work sharing. Hence, as the number of source nodes increases, it becomes more expensive to use the left-recursive version. To illustrate, even though node *f* is reachable from both nodes *b* and *e*, these two nodes have to separately infer their reachability facts to *f* and nodes beyond. As a result, when node *e* is included as a source node together with node *b*, the number of messages increases to 16. It becomes 21 messages when node *g* is included.

4.1.3 Reducing the Number of Iterations with the Smart Algorithm

In all our previous examples, we have discussed execution strategies that produces new reachable facts *one-hop-at-a-time*. These strategies utilize *semi-naive evaluation* [3], which ensures that the *fixpoint* is reached with a number of iterations bounded by the length of the longest path in the queried graph.

As an alternative, the *Smart Algorithm* [17, 9] reduces the number of iterations by computes paths of length 1, 2, 4, 8, etc and hence reduce the number of iterations to logarithmic the length of the longest path. Figure 14 shows the query plan for the *Smart Algorithm*. In relational algebra, the *Smart Algorithm* is expressed as follows:

```
$delta := link$
$reachable := link$
repeat
    $delta := delta \bowtie_{delta.Y=delta.X}\ delta$
    $power := delta \bowtie_{delta.Y=reachable.X}\ reachable$
    $reachable := reachable \cup delta \cup power$
until $delta = \emptyset$
```

In this algorithm, *delta* facts produced have path length that are powers of 2, i.e. 1,2,4,8, etc at each increasing iteration. *Power* produces reachability facts that are of path length 1 at the 0^{th} iteration, 3 at the 1^{st} iteration, $5-7$ at the 2^{nd} iteration, and $9-15$ at the 3^{th} iteration. Taking the union of *power* and *delta* facts would generate all possible *reachable* facts.

However, while the *Smart Algorithm* reduces the number of iterations, it incurs higher communication overhead due to the extra join being performed. On top of that, it also introduces duplicates as an undesired side-effect. For example, *delta* facts of path length 3 may be generated from *delta* facts of path length 1 and 2, even though *delta* facts should only have path length that are powers of 2. Similarly, *reachable* facts can be generated in multiple ways: by joining a 4-hop *delta* and a 5-hop *reachable* tuple, or a 8-hop *delta* and a 1-hop *reachable* tuple.

To prevent duplicate facts from being produced, all *delta*, *power* and *reachable* facts contain an extra *hop* field, which keeps track of which iteration the fact is produced. *delta* facts are joined with other *delta* facts produced in the same iteration. Similarly, *reachable* facts are joined with *delta* facts produced in later iterations.

Note that the query plan in Figure 14 can be further opti-

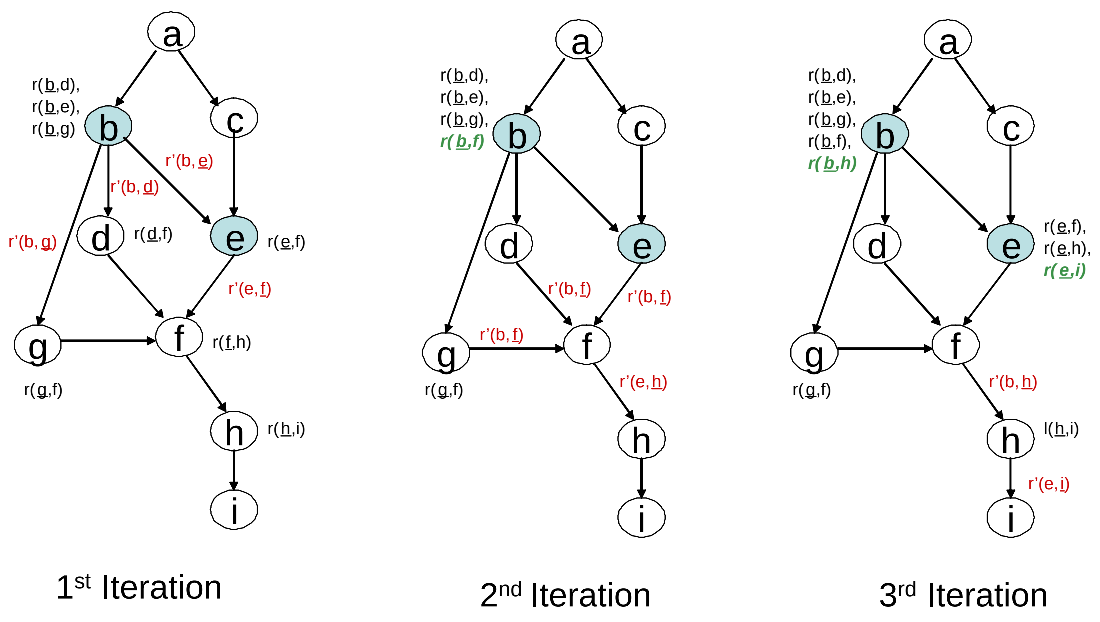


Figure 13: *The communication patterns for executing query in Figure 12.*

mized by making the observation that the *delta* facts are being rehashed twice by their destination fields for each of the join. We can merge these two rehashes into one to further reduce communication overhead. This technique is also discussed earlier in Section 4.1.1 and used for optimizing the query plan in Figure 9.

4.1.4 Detecting Cycles

A cycle in the queried graph can lead to a potential infinite loop during query execution. When semi-naive evaluation techniques are utilized, as new *reachable* facts are discovered "hop-at-a-time", detecting cycles can be trivially done by comparing source and destination fields in all *reachable* facts.

When using the *Smart Algorithm*, cycles become harder to detect. Consider a cycle of 5 nodes: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow a$. Because the number of hops increases by a factor of two, *delta* facts that are produced by the execution plan in Figure 14 would not always be able to detect the 5-hop cycle. For example, a 8-hop *delta* fact would "wrap around" the cycle and end up having a *delta(a,c)* fact that contains a cycle $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow a \rightarrow b \rightarrow c$.

To prevent generating such redundant *delta* facts that contain cycles, we currently maintain an additional *path* field that stores the entire path for each *delta* and *reachable* fact. Cycles are then detected by checking the *path* field. Storing this extra field increases communication overhead but would be offset by the decrease in redundant *delta* facts.

4.2 Queries with Aggregation

In most of our applications, queries typically compute aggregates in addition to transitive closure. For example, these queries can compute all-pairs shortest paths, number of paths between any two nodes or the diameter of the network. In our implementation, aggregates can be produced either *periodically*, or *incrementally* (as the aggregate value changes). Incremental computation works only for monotonic aggregates.

We revisit the *Shortest Path* program for computing the shortest path from node *a* to *f*. This time, we apply magic sets optimization described in Section 4.1.1 to limit the computation to only nodes reachable from node *a*:

R10: *shortestPath(X,Y,P,C) :- shortestLength(X,Y,C), path(X,Y,P,C).*
R11: *path(X,Y,P,C) :- magicNodes(X), link(X,Z,C_2), path(Z,Y,P_1,C_1), P = addLink(link(X,Z),P_1), C = C_1 + C_2.*
R12: *path(X,Y,P,C) :- magicNodes(X), link(X,Y,C), P = addLink(link(X,Y), nil).*
R13: *shortestLength(X,Y,min<C>) :- path(X,Y,P,C).*
R14: *magicNodes(Y) :- magicNodes(X), link(X,Y,C)*
R15: *magicNodes(a)*
Query: *?- shortestPath(a,f,P,C).*

While the above query computes the shortest path from node *a* to node *f*, as a side-effect, the query also computes the shortest path to all nodes reachable from *a*. The query is inefficient because it enumerates all possible paths from node *a*. We can avoid enumerating all possible paths with the use of aggregate selections [16].

To illustrate, consider the graph in Figure 15 where there are three different paths from node *b* to node *f*. Only the path $b \rightarrow g \rightarrow f$ is relevant to node *a's* computation of the shortest path. We can avoid traversing the other longer paths by maintaining an aggregate value for the current shortest path cost from node *b* to node *f*. This value is stored by the source node and can be used to "prune" away any paths that exceeds this value. The use of aggregate selections is possible only for monotonic aggregations. This also suggests that by reordering the link tuples being used in the computation, we may be able to compute

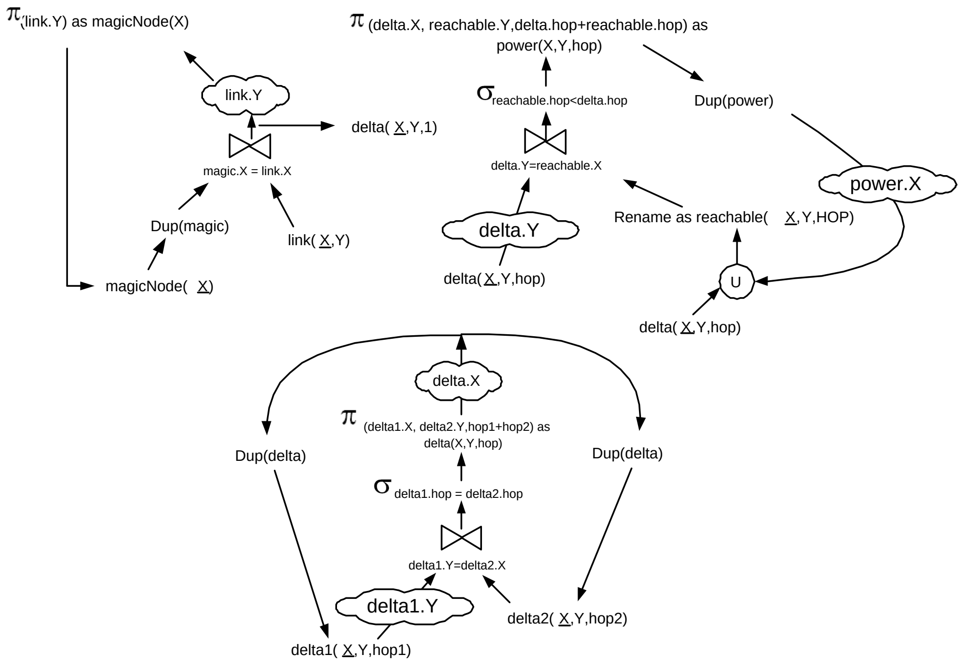


Figure 14: *Query Execution Plan for Smart Evaluation of the Magic Reachable program.*

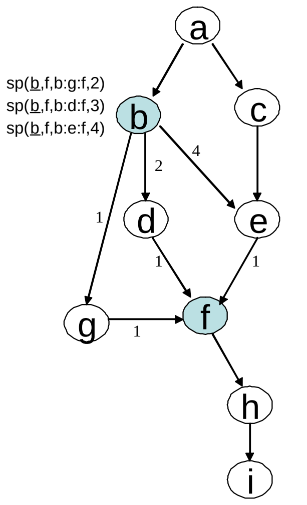


Figure 15: *All Possible Paths from node b to node f.*

the path $b \rightarrow g \rightarrow f$ before the other paths and hence prune the other edges, leading to savings in communication costs.

Aggregate selections can easily be adapted to compute *shortest-k* paths by keeping the the min-k values of the path length instead of just the shortest path. We can also support a combination of the two, where different source nodes have different requirements on the number of shortest paths it wishes to compute.

Note that if we are computing the *Route Resilience* query to find all possible paths from node *b* to node *i*, the aggregate selection is not applicable.

The query plan to compute all-pairs shortest path is shown in Figure 16. This plan utilizes the aggregate selection operator for the enhancement described above. If we are interested in computing the diameter of the network, a *max* aggregate can be applied to all *shortest_path_length* facts. The maximum value is the diameter of the network. In this plan, the *shortest_path_length* and *path* facts are both keyed by their source nodes to avoid having to incur extra communication when joining these two by the source and destination fields. The choice of what key to index these generated facts is done manually at this moment. We hope to automate this decision on intermediate data placement into the query optimizer in future.

Note that our query plan performs the aggregation *within* the recursion itself. For example, the shortest path from node *d* to node *i* is stored on node *d* and used to compute the shortest path from node *b* to node *i*. This enables work sharing to occur. The other alternative would be for node *b* and node *d* to compute their own separate shortest paths using DHT-based hierarchical aggregation techniques.

5 Querying Dynamic Graphs

In practice, the monitoring queries we are executing over the queried graph will be long running continuous queries. In this section, we will provide a brief description of how long-running queries over dynamic graphs can be supported. Note that the queried graph itself is maintained as *soft-state*, i.e. the individual $link$ facts are never explicitly deleted but rather timeout or renewed by the application.

One convenient way to maintain the state on these long-running queries is to view them as *materialized views*, where intermediate query results and computation states are stored in

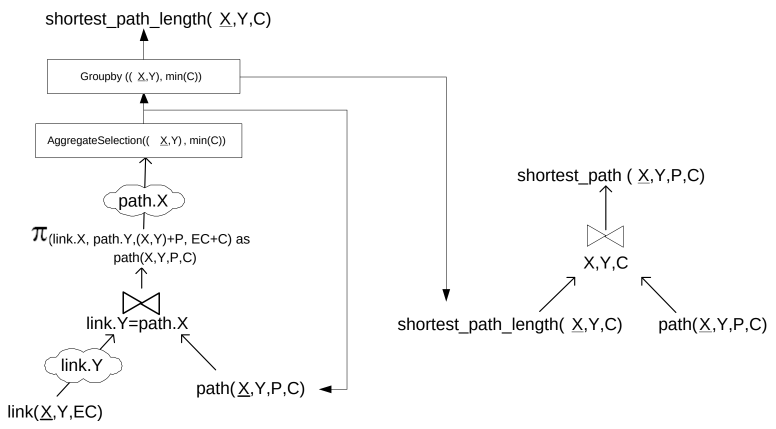

shortest_path_length(X,Y,C)
Groupby ((X,Y), min(C))
AggregateSelection((X,Y) , min(C))
path.X
π(link.X, path.Y,(X,Y)+P, EC+C) as path(X,Y,P,C)
link.Y=path.X
link.Y
link(X,Y,EC)
path(X,Y,P,C)
shortest_path (X,Y,P,C)
X,Y,C
shortest_path_length(X,Y,C)
path(X,Y,P,C)

Figure 16: *Shortest Path Execution Plan with Aggregate Selection*

the DHT. On top of supporting *long running queries*, materializing the views in the DHT can be used for the following purposes:

- **Inter-Query Work Sharing:** Earlier, we have shown that work sharing within a query is possible to avoid traversing the same path multiple times. Work sharing across queries can be done across queries. For example, if a shortest path $b \rightarrow d \rightarrow f$ has been computed by one query, it can be used by another query to find the shortest path $a \rightarrow b \rightarrow d \rightarrow f$.
- **Real-time Query Support:** In the applications described earlier, there are *real-time* application queries that would benefit from precomputed query results on graph topology. First, routing between two nodes or directed flood query towards higher degree nodes would require up-to-date information on node-degrees of the Gnutella network. This information must also be readily available when a search query is issued. Second, in the routing infrastructure application described in Section 3.4, the all-pairs shortest path can be precomputed and maintained as as a materialized view in the DHT. This information is then shared by all route formation queries.

As the network graph evolves as nodes enter and leave the system, the materialized view needs to be incrementally updated. Unlike a non-recursive view, addition or deletion of a single base tuple may cause several updates to occur recursively on one or more views. For example, loss or addition of a link may mean updating several reachable tuples.

The approaches taken previously for maintaining updated views involve issuing an incremental query *Q"* to modify view. For example, the *DRed* (Delete and Rederive) [7] scheme that first computes an over-estimation of the deleted tuples, and then regenerates missing tuples. Such techniques do not work when there are no explicit deletes and all data on the queried graph is maintained as soft-state.

We will instead make use of *timestamps* and queries with *window semantics*. We consider new links and nodes being published as a distributed stream of tuples, each being timestamped with their creation time and a lifetime. Derived facts from the *Reachable*, *Shortest Path* programs are timestamped based on the *oldest* timestamp of the base tuples used to form the query. Queries can be issued to only compute new facts using base tuples that fall within a window period of time. Existing operators such as duplicate elimination, aggregate selections have to take into account the windows semantics.

6 Experimentation

We present experimental results by running two queries: the *Reachable* and *Diameter* queries described in Section 4.1. The queries are executed using PIER running over Chord [15] on a network simulator of 100 nodes. The latency is between any two nodes is set to 100ms and the bandwidth is set to 1.5Mbps. We simulate congestion at each node by imposing FIFO queues on messages being sent.

Our experiments are based on the *external network queries*, where PIER nodes are used to query an external graph. The *link* facts of external graph is first published into the DHT, and queries are issued over this graph. Our experiments focus on the following three metrics:

- **Results Latency:** The arrival time of results as they are generated and stored in the DHT. In our experiment, all result tuples generated are sent back to their query node, which then timestamps their arrival time relative to the query issue. The fixpoint latency defined in Section 4 is set to the arrival time of the last result tuple.
- **Rehash Overhead:** The communication overhead, measured by the total message size in MB of tuples rehashed during query execution.

Our experiments are conducted on directed graph size of 100 and 500, and we will show that the same results hold for undirected graphs in Section 6.4. For the smaller graph (size=100), we vary the average node degrees (ANDegree) 1, 2 and 4, with network diameters of 21, 12 and 9 respectively. For the larger graph (size=500), the respective network diameters are 61, 22 and 15. A larger *ANDegree* indicates a denser network.

6.1 Transitive Closure with Source Limitations

In our first experiment, we compare different execution strategies by varying the *source selectivity (SS)*, which is the fraction of source nodes in the *Reachable* program. The larger the *SS*, the greater the portion of the graph being queried. The three strategies being compared are *Right Recursion (RR-Full)*, *Magic Right Recursion (MRR-Partial)* and *Left Recursion (LR)*. Their respective query plans are shown in Figures 6, 9 and 12. We added a "Full" or "Partial" to the strategies, where "Full" means the query execution will require a full transitive closure even whether the query is limited to a number of source nodes. Both *LR-Full* and *LR-Partial* uses the same query plan in Figure 12. The only distinction being *LR-Full* does not have the selection predicate to filter by source nodes (hence resulting in a smaller query plan as the set of source nodes can be arbitrarily large).

Figures 17, 18, 19 show the rehash overhead (MB) as *SS* increases for all three approaches for *ANDegree* values of 1, 2 and 4 respectively. Figures 20, 21 and 22 show the same graphs for *fixpoint latency(s)*. We make the following observations:

- As *ANDegree* increases, the graph contains more *link* facts and this leads to an increase in rehash overhead. However, the increase in the number of links leads to a smaller network diameter which reduces the fixpoint latency.

- *MRR-Partial* is effective for reducing the rehash overhead for sparse graphs and low *SS*. For example, when *ANDegree*=1 and *SS*=0.01, the reduction in communication overhead is 77%. As *SS* increases, the rehash overhead for *MRR-Partial* increases. However, the increase is not linear due to the benefits obtained from work sharing as paths overlap. I.e., each additional source node increases the overall rehash overhead by a diminishing amount. In the extreme case, it will be equivalent to the total cost of computing full transitive closure using right recursion and the cost of publishing the initial magic nodes. As the network becomes denser (*ANDegree* increases), magic sets becomes less effective in reducing rehash overhead because the number of reachable nodes becomes larger on a denser graph even when *SS* is low.

- *MRR-Partial* incurs slightly higher fixpoint latency compared to *RR-Full* for most values of *SS*. This is attributed to the increased number of iterations required to reach fixpoint as a result of using magic sets. The only anomaly happens when *ANDegree*=1 and *SS*=0.01 where the queried portion of the graph is so small that the number of iterations is fewer than the diameter of the network.

- *LR-Partial* is the most bandwidth efficient when there are not many path overlaps between different source nodes. Our results show that *LR-Partial* incurs the least rehash overhead when *SS* is below 0.3, 0.6 and 0.8 for *ANDegree* values of 1, 2 and 4 respectively. As *SS* increases, *LR-Partial* shows linear increase in rehash overhead. This linear increase is due to the fact there is no work sharing involved even when paths between reachable nodes overlap. Since our graphs are randomly generated, the addition of a source node causes a linear increase to the number of rehashes required. In the worse case (*SS*=1), the rehash overhead is the same as *LR-Full*, and twice that of *RR-Full*.

- *LR-Full* incurs roughly the same number of iterations as *RR-Full*, and hence the fixpoint latency is the same for the two approaches. This suggests that latency, rather than bandwidth is the bottleneck, since *LR-Full* requires twice as much bandwidth compared to *RR-Full*. However, for *LR-Partial*, the fixpoint latency is slightly higher than that *RR-Full*, especially for large values of *SS*. We attribute this to overheads in query dissemination, where *LR-Partial* has to store the set of all source nodes, which grows as *SS* increases.

We repeated the experiment for graph size of 500. We omitted the graphs for brevity, but they show similar trends when comparing across different schemes. In absolute values, the rehash overheads and fixpoint latency values are higher due to the larger graph sizes.

6.2 Smart Algorithm

Next, we evaluate the effectiveness of the *Smart Algorithm* which reduces the number of iterations, at the expense of generating more communication overhead. We compared the results latency of this algorithm against two semi-naive approaches examined earlier. Figures 23, 24 and 25 shows the results time series for computing the full transitive closure queried graph size of 100. The Y-axis shows the cumulative number of results obtained after a time lag from query issue (shown on X-axis). The three strategies compared are two semi-naive approaches *LR-Full*, *RR-Full* from the previous experiment, and the *Smart Algorithm*, shown as SA-Full on the figures.

On sparse graphs with large diameter (*ANDegree*=1), *SA-Full* has the lowest latency. The fixpoint latency of *SA-Full* is 34% lower than that of RR-Full, while generating $2.3\times$ as much traffic. This is indicative that reducing the number of iterations has an impact on reducing the fixpoint latency.

However, as the graph becomes denser with reduced diameter (*ANDegree*=2), *SA-Full* performs almost as well as semi-naive approaches, receiving all but a small number of results within 12.8 seconds, which is the fixpoint latency of *RR-Full*. There is the presence of a "long-tail", which indicates bandwidth bottlenecks caused by hotspots on certain nodes. We will investigate the causes of the hotspots in future.

When (*ANDegree*=4), the differences in performance between *SA-Full* and the semi-naive approaches is even more apparent. Overall, our results shows that while *SA-Full* can reduce the number of iterations, it is not effective in reducing the latency of dense graphs where the number of iterations is small to begin with. Further, the excess communication overhead generate may create a negative impact on results latency.

6.3 Transitive Closure with Aggregation

In this experiment, we examine the performance of a transitive closure query with aggregation: the *Diameter* query which computes all-pairs shortest path lengths described in Section 4.2, and each node incrementally sends the maximum shortest path value encountered to the query node.

On top of measuring fixpoint latency, we also measure the *Convergence Latency*, which is the time lag from query issue until the final diameter value is received. Convergence Latency provides an indication on how quickly the system converges on an accurate value of the network diameter.

Table 1 summarizes our experimental results. As ANDegree increases, the rehash overhead increases, while fixpoint latency decreases. This is consistent with our earlier observations for computing transitive closure. There are two other observations we make. First, the use of aggregate selections can improve the

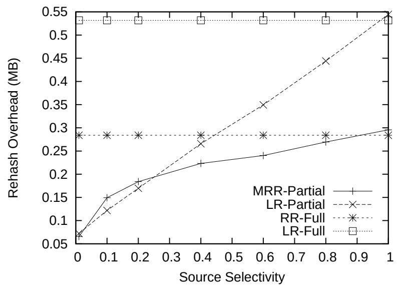


Figure 17: *Rehash Overheads (size=100, ANDegree=1)*

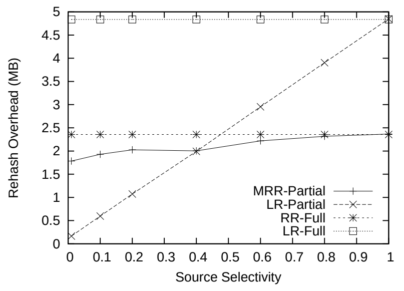


Figure 18: *Rehash Overheads (size00, ANDegree=2)*

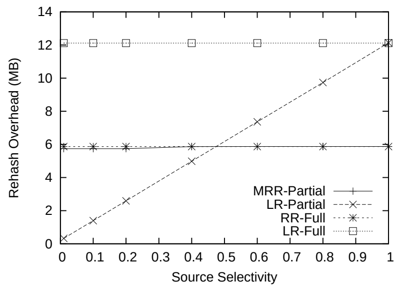


Figure 19: *Rehash Overheads (size=100, ANDegree=4)*

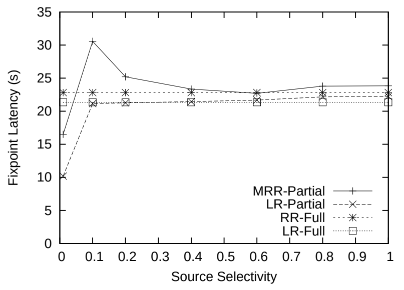


Figure 20: *Fixpoint Latency (size=100, ANDegree=1)*

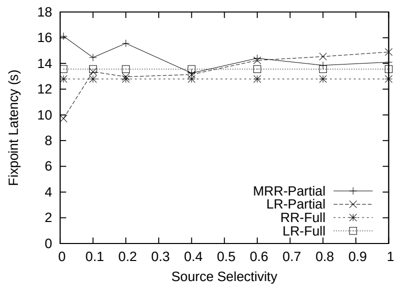


Figure 21: *Fixpoint Latency (size=100, ANDegree=2)*

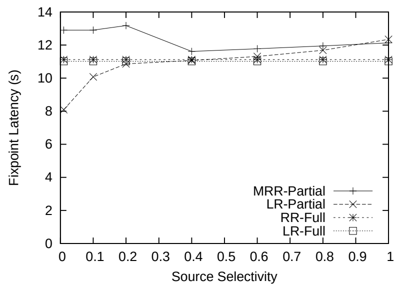


Figure 22: *Fixpoint Latency (size=100, ANDegree=4)*

fixpoint latency significantly, without compromising the Convergence Latency. When ANDegree=1, there is only one path between any two nodes and hence the use of aggregate selections is irrelevant. However, when ANDegree increases to 2, by avoiding enumerating all possible paths, aggregate selections reduces the fixpoint latency from greater than 100s to 12.9s, at least a 7$\times$ reduction. In fact, the fixpoint latency of computing the diameter is even lower than that for computing the full-transitive closure for the *Reachable* query (14.1s) that performs duplicate elimination based on (source,destination) pairs. The reduction is even more apparent as ANDegree increases as there are more paths from which to "prune".

Our second observation is that as ANDegree increases, the *accurate diameter latency* becomes much lower than the *fixpoint latency*. This is a direct result of the graph density itself. A sparse graph would require computing more shortest-path information before the diameter (maximum shortest-path) is computed. A dense graph would compute the diameter value much faster. This suggests that for a dense graph, sampling a set of source nodes may yield a close estimation of the actual diameter. We will explore the use of such sampling techniques.

In future, we will examine the effects of reordering the links for computation based on their link cost to quantify the reductions to latency. We believe a combination of aggregate selections and reordering would further reduce the latency and communication overhead.

6.4 Undirected Graphs

We repeated our experiments on undirected graphs. We impose no restrictions on path costs or hop count, and as a result, there is no difference between computing full or partial transitive closure. We experiment with graph size of 100 and 500, for ANDegrees of 2 and 4. An ANDegree of 2 means an undirected edge between two nodes (one incoming, and outgoing). Our experimental results confirm that the relative differences across the three schemes (*RR-Full*, *LR-Full* and *SA-Full*) are similar to that obtained from directed graphs for full transitive closure computation. We omit the resulting graphs for brevity. In future, we will compare the different schemes for partial transitive closure by limiting each source node to reachable nodes within a fix number of hops or path cost.

6.5 Summary of Results

We summarize our experimental results as follows:

- Right recursive transitive closure have the desired effect of sharing work within the same query whenever there are path overlaps. However, when the query is limited to a small set of source nodes, even with the use of magic sets, right recursion will result in the computation of redundant facts not required by the query. Applying magic sets rewrite to the right recursive transitive closure reduces communication overhead, and the reductions is most significant when the number of source nodes is small and the network graph is sparse. They are not effective in reducing communication overhead for dense graphs. However, magic sets is still a preferred option compared to not applying magic sets for the following two reason. First, in the worse case when all nodes in the network are reachable from the source nodes, using magic sets is only slightly more communication overhead compared computing the full transitive closure without the use of magic sets. If the graph topology is not known in advance, it almost certainly pays off in bandwidth savings to perform magic sets rewriting. Second, while the use of magic sets degrade

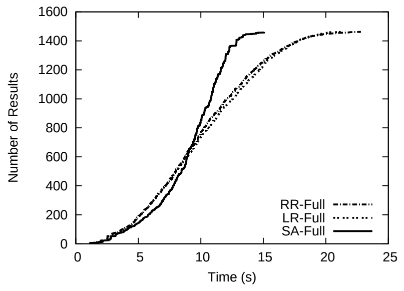


Figure 23: *Results Time Series (size=100, ANDegree=1)*

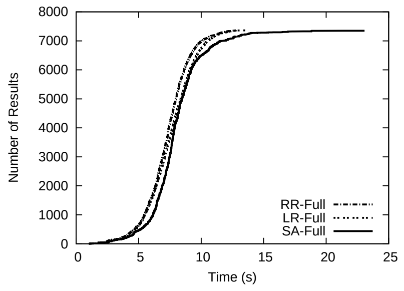


Figure 24: *Results Time Series (size=100, ANDegree=2)*

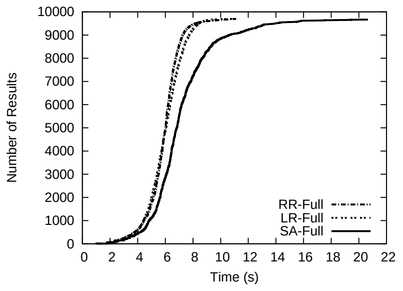


Figure 25: *Results Time Series (gs=100, ANDegree=4)*

ANDegree	Rehash Overhead (MB)	Fixpoint Latency (s)	Convergence Latency (s)
1	0.33	22.8	22.03
2	3.19	12.9	11.3
4	9.97	11.2	6.8

Table 1: Results for *Diameter* query

results latency, the difference is marginal and tolerable in most cases.

- Left recursive transitive closure results in significant savings in rehash overhead when the source selectivities is low. However, because this technique does not take advantage of work sharing when there are path overlaps between source nodes, the rehash overhead increases linearly as the number of source nodes increases. This results in high bandwidth consumption as source selectivity increases. In our experiments, when full transitive closure is performed with left recursion, the bandwidth consumption was twice that of right recursion. While left recursion reaches a fixpoint in the same number of iterations as right recursion, the increase in bandwidth consumption would have a negative impact on large and dense graphs.
- The *Smart Algorithm* has the desired effect of reducing results latency when the network is sparse and the network diameter is large. However, for a dense graph with low network diameter, the benefits derived from a reduction in the number of iterations can be offset by the network delays caused by the extra communication overhead.
- The use of aggregate selections can improve the performance of computing all-pairs-shortest path. As the queried graph becomes dense, the improvements will be even greater as there are more avenues for pruning unnecessary paths.

7 Future Work

As part of our future work, we would like to explore some of the following:

- **Queries over Dynamic Graphs:** In Section 5, we discussed how queries over dynamic graphs can be supported. Our experiments have only dealt with queries over static graphs. We want to experiment with long running queries over dynamic graphs, and to have a better the effects of different strategies on achieving fixpoint efficiently when the graph is dynamic. We also hope to define more accurately what a fixpoint means within the context of a dynamic graph.
- **Query Dissemination:** In our implementation, the query is being disseminated to all nodes. In our experiments, since the query duration and the queried graph size is small, the cost of query dissemination turns out to be fairly significant relative to rehash overhead. When computing partial transitive closure, only nodes that are reachable from the source nodes need to receive the query. In future, we will support query dissemination where the query is "piggy-backed" with data and disseminated as new facts are generated to be processed on nodes that have not received the query. However, for long-running queries, it may be alright to broadcast the query to all nodes since the initial cost of disseminating the query is insignificant relative to the communication overhead incurred by the long-running query.
- **Transformation of Datalog to Execution Plan:** Currently, we do not have an optimizer implemented, and all of our query plans are "hand-built" using datalog diagrams. Figure 5 best summarizes the various options for transforming a declarative Datalog query to optimized execution plan. We have shown that depending on the query workload and graph density (indication of join selectivity), the query execution plan can make a significant difference to both performance and system overhead. We would like to explore automating this transformation from a declarative plan to an optimal execution plan.
- **Adaptive Query Optimization:** After producing the "optimal" query plan, given that the network is dynamic and our queries are long running, adaptive query optimization techniques such as Eddies [1] may have an important role in this context. Our experimental results suggests that a hybrid approach that utilizes left and right recursion appropriately depending on the graph query ,

- **Applications:** We are in the process of prototyping the Gnutella monitoring application. We will also be planning to deploy the other applications discussed in Section 3.

8 Conclusion

In this paper, we examine the use of recursive queries for querying network graphs. We survey a variety of applications where these queries would be used for both analyzing networks and for routing. The distributed, dynamic and large-scale nature of these networks motivate the use of *in-network* query processing techniques. We propose the use of PIER, a P2P query engine to execute these queries.

To understand the performance characteristics of executing recursive queries in distributed multi-hop networks, we focus our study on the transitive closure query, using the canonical *Reachable* program. We examine the communication patterns observed in running the transitive closure query over the DHT, comparing left and right recursion techniques, as well as semi-naive vs squaring approaches. We also discuss the *Shortest Path* and *Diameter* queries, and motivate the use of aggregate selections can reduce communication overhead.

Our experimental results also show that the depending on the query and graph topology, the choice of the best execution strategy can lead to significant improvements in communication overhead and latency. Section 6.5 summaries our experimental findings.

9 Acknowledgements

The authors would like to thank Joseph Hellerstein, Sridhar Rajagopalan, Raghu Ramakrishnan, Ion Stoica, and members of the PIER research group and Berkeley Database group for their insights and suggestions.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems.

[3] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, 1987.

[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] Boon Thau Loo and Sailesh Krishnamurthy and Owen Cooper. Distributed Web Crawling over DHTs. *UC Berkeley Technical Report UCB//CSD-4-1305*, Feb 2004.

[6] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the ACM SIGCOMM 2003, Karlsruhe*, 2003.

[7] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[8] 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.

[9] Y. Ioannidis. On the computation of the transitive closure of relational operators. 1987.

[10] Karthik Lakshminarayanan and Ion Stoica and Scott Shenker. Building a Flexible and Efficient Routing Infrastructure: Need and Challenges. (UCB/CSD-01-1141), Apr. 2001.

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

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

[13] S. Rhea, T. Roscoe, and J. Kubiatowicz. Structured Peer-to-Peer Overlays Need Application-Driven Benchmarks. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[14] S. Sizov, M. Biwer, J. Graupmann, S. Siersdorfer, M. Theobald, G. Weikum, and P. Zimmer. The bingo! system for information portal generation and expert web search.

[15] 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.

[16] 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.

[17] P. Valduriez and H. BORAL. Evaluation of recursive queries using join indices. In *Proceedings of the 1st International Conference on Expert Database Systems*, 1986.