Processing Queries in a Large Peer-to-Peer System

Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt

University of Wisconsin-Madison Computer Sciences Department
1210 West Dayton Street, Madison, WI 53706, USA
{lgalanis, yuanwang, jeffery, dewitt}@cs.wisc.edu

Abstract. While current search engines seem to easily handle the size of the data available on the Internet, they cannot provide fresh results. The most up-to-date data always resides on the data sources. Efficiently interconnecting data providers, however, is not an easy problem. Peer-to-peer computing is the latest technology to address this problem. However, efficient query processing in peer-to-peer networks remains an open research area. In this paper, we present a performance study of a system that facilitates efficient searches of large numbers of independent data providers on the Internet. In our scenario, each data provider becomes an autonomous node in a large peer-to-peer system. Using small indices on each node, we can efficiently direct queries submitted on any node to the relevant sources. Experiments with a large peer-to-peer network demonstrate the feasibility of our approach.

1 Introduction

Information on the Internet is constantly growing and changing. Search engines and current web repositories ([9], [10] and [16]) can efficiently handle the amount of data on the web and provide fast searches over the data they have crawled and indexed. Nevertheless, they cannot provide the most up-to-date results or search the data storage back-ends of sites that dynamically generate web pages. The only way to access the most recent versions of available data is to directly submit searches to each data source. Therefore, it is not surprising that most content publishers provide their own search service.

A single web query may require a very large number of sites to provide results. Hence, a system is needed that is able to efficiently identify the relevant data sources. A candidate approach would be to organize all the data sources in a large peer-to-peer (P2P) network where queries are answered by the relevant sites. Such a distributed system has several advantages. There is no central point of failure and no central repository necessary to facilitate searches. The absence of a global authority allows sites to enter and leave the system at will, as well as independently manage their data. Finally, each query will always retrieve the most up-to-date results.

Deployment of such a P2P system allows efficient searches across the entire content of a large number of sites. To achieve this, the system must exhibit certain qualities. For each query, it is necessary to efficiently locate all the relevant data sources. Additionally, the entire system must scale to a large number of sites without compromising site independence or performance.

Motivated by the emerging need to query data distributed across many independent data sources, we have built a prototype distributed query processing system for que-

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 273–288, 2003.

ries on XML data. Our system makes use of data items that change infrequently and often appear in queries, such as metadata and words characteristic of a specific site. Indices over these data items map this data to the corresponding sites and use these indices to direct queries. When sites join the system, they exchange information that allows for the construction of these indices. Thus, query content drives selection of promising data sources.

Our prototype distributed search engine, the low-level layer of the query processing system, was used to run experiments using real data on a test bed consisting of up to 200 peers. We studied system performance with various levels of information exchanged among nodes in the system. The amounts of exchanged information ranged from no information, which is similar to a Gnutella-like ([8]) processing strategy, to full summary information replicated on each node. Our experiments show a larger than expected performance increase as the number of messages passed at query time is reduced. As more information about peers in the network is stored on each node, network bandwidth usage is reduced due to a large reduction in the number of messages exchanged. Consequently, when each node stores complete summary information about all peers in the P2P network, the average query response time drops by a factor of as much as 75 and query throughput improves by as much as a factor of 20 over a Gnutella-like approach. Our approach efficiently connects data providers, allows queries to retrieve the most up-to-date data, preserves site autonomy, and puts reasonable demands on the bandwidth of the network.

This paper is organized as follows: In Section 2, we present the system architecture of our approach. Section 3 describes the join policies for peers. Section 4 outlines the methods for selecting peers relevant to queries. Section 5 describes our experimental setup. We present the results of our experiments in Section 6. Section 7 reviews related work and Section 8 draws conclusions and presents future research directions.

2 System Architecture

Each node in the peer-to-peer system consists of an XML search engine and a communication infrastructure that allows communication among peers. The local XML search engine [29] enables document searches based on both structure and content. Along with the local indices, each node maintains additional indices that facilitate the forwarding of queries to data providers that have relevant data. When peers join the peer-to-peer system, they exchange information with peers already in the network for two main reasons: a) They need to make their presence known to the system and advertise their data and b) they want to obtain index information about other peers in the system from the peers to which they are directly connected.

Advertising a node's data to other nodes in the system requires a meaningful summarization of gigabytes of data. Fortunately, XML provides mark-up tags that describe a data collection. Usually, a large XML data collection has a relatively small set of descriptive XML elements. Furthermore, a data collection at a node may have text words that are characteristic of that node. We will refer to XML elements and text words collectively as *keywords*. For example, an art auction site can be distinguished by the fact that it stores elements called *item*, *price,* or *current_bid* and contains text words such as "painting" or "sculpture". The elements for an online retailer could be *item*, *price*, and *brand.* It would also have different characteristic text words

such as “electronics” or “books”. XML metadata and classification techniques such as [12] and [3] provide further ways to summarize the content of peers. Throughout this paper it is assumed that the XML tags provided by the participating nodes have-system wide defined semantics. The system does not try to compensate for semantic heterogeneity, which is an orthogonal problem. Section 4 outlines why our design is still valid even in the presence of semantic heterogeneity.

2.1 XML Search Engine

Our search engine framework supports the execution of *containment queries* that fully exploit the structure in XML documents. For example, the query (in XPath notation [25]) *//book//author//name=”John Smith”* returns all documents that contain books by John Smith. Each search engine indexes its local data using inverted lists, which map keywords to documents. These indices enable the processing of containment queries as in the Niagara system [14]. Our version uses B+-Trees to implement inverted lists and employs an improved version of the Multi-Predicate Merge Join algorithm presented in [29]. A search engine that facilitates complex queries over structured data distinguishes our system from file-sharing systems.

2.2 Peer Indices

In addition to its local inverted indices, each node maintains indices containing information residing at other peers in the network. The *peer index* (PI) is a simple inverted list that maps keywords to peers in the network and is used for selecting peers based on query content. Each peer stores in its PI all keywords published by other peers during joining. Using the PI, a query can be sent directly to nodes that are likely to have results. Section 4 presents how our prototype system uses the peer indices.

The number of keywords in the PI, and thus the size of the PI, depends on how many keywords each peer decides to publish when it joins the system and how many keywords each peer decides to retain in its PI. We expect a node to only publish a small subset of its keywords, such as words that are often found in queries or that are especially representative of its local data. XML tags are especially good candidate keywords since they are regarded as being descriptive of the actual data. PI entries for tags can also contain structural information which allows the system to further distinguish among data sources.

The simplicity and generality of the PI concept makes it applicable to data collections other than XML. We chose XML for our prototype system because it seems to be the emerging standard for information exchange and data integration. In general, any distributed data collection that can be queried using some query language can benefit from PIs. The rough guideline is as follows: Items that appear in queries and are characteristic of a data collection should be put in the PI. For example in the context of relational data one could put attribute and table names in the PI in order to facilitate query routing given a SQL query. Furthermore, selected data values (such as names of cities) can also be incorporated into the PI.

2.3 Horizons

Knowledge of all the peers in the system may seem limiting for scaling the system to a very large number of peers. By introducing a *horizon*, a node can bound the number of peer IDs that it stores in its PI.

Definition 1: A node *N* enforces a *horizon* if it stores in its PI only a subset of the peers in the network. IDs of peers outside of *N*'s horizon are substituted with IDs of peers within *N*'s horizon. Queries that need to be sent to peers outside *N*'s horizon are relayed by nodes within *N*'s horizon.

When a peer joins the P2P network, it starts with an initial PI obtained from another peer and then modifies its PI based on local storage constraints and query workload characteristics, evicting infrequently used keywords and nodes. Thus, a peer's horizon evolves to contain information about frequently queried keywords and nodes that usually return relevant results. The evolution of a PI horizon is depicted in Fig. 1. The size and shape of the horizon is dependent on *N*'s storage constraints and query workload. The tradeoff is that a small horizon will result in more messages for a query that retrieves data outside the horizon, whereas a large horizon will require more storage space for peer information.

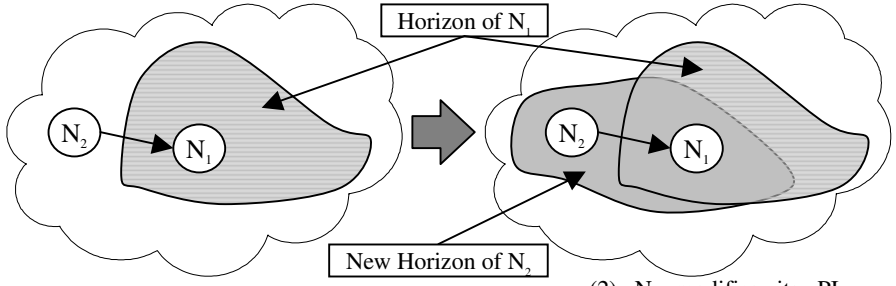


Fig. 1. The evolution of a peer index

3 System Evolution

A peer becomes part of the P2P network by *joining*. The general case of a join between two nodes, shown in Fig. 2, arises when node N_1 contacts node N_2 by sending a *join request*. Both nodes have already joined with other nodes and belong to existing join graphs.

Definition 2: A *Join Graph* is a non-directed graph $G(V, E)$ where $V = \{N_i \mid \text{node } N_i \text{ has joined with at least one } N_j \neq N_i\}$ and $E = \{(N_i, N_j) \mid \text{node } N_i \text{ has joined with } N_j\}$.

Definition 3: The *Distance* of two nodes N_i and N_j ($d(N_i, N_j)$) is the number of hops on the shortest path from N_i to N_j in the Join Graph.

The join graph represents a logical overlay network and illustrates how nodes joined with one another. It does not correspond to the underlying network topology. Thus, the distance between two nodes does not necessarily reflect the communication cost between them. The system uses the join graph for every message that has to flow to others peers in the system, such as join update messages and update messages that nodes create when their data changes.

When peers join the system, they receive information about already existing peers. Thus, a newly joined node ends up with a PI that contains information about all exchanged keywords in the system and all the peers in its initial horizon. Joining starts when a peer N_1 contacts another peer N_2 and requests to join the P2P network. The joining nodes exchange messages containing information about their local data and their peer index. This new information is then sent to each node's neighbors. The resulting messages flow throughout the join graph and are processed exactly once on each node. Thus, a join between two peers creates a wave of messages that eventually reaches all the peers in the system. The result is a peer-to-peer network in which each node has a summary of the data on all other peers.

Propagating information to every node in the network whenever a new node joins may seem limiting for the network's scalability. We, however, expect our system to be used by data repositories that want to make their data available as long as possible with only short down-times, rather than by individual users that share media files. It is expected that a peer may go down for a short time. When it comes back up, however, only an update containing the returning node's changed data, if any, needs to be propagated through the network.

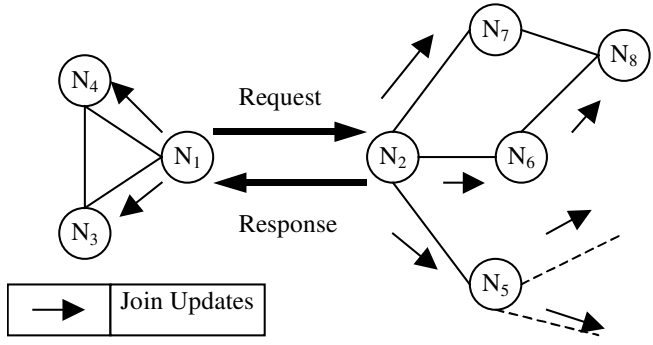


Fig. 2. Node N_1 joins with node N_2

We employ a simple network join algorithm that guarantees all nodes eventually obtain a summary of the data on all other peers. However, if nodes do join and leave frequently, alternative methods of joining can be employed. One such approach would be to use an *incremental join*. As with the simple join algorithm, the incremental join begins with a newly arriving peer receiving a PI from a directly connected peer. However, update messages about the new peer are not propagated to all neighbors. As the new peer issues queries, it piggybacks selected keyword data on its query messages. The data that is attached to the message is based on factors such as the content of the query, the length of time the node has been connected, and the size of the message. Through this process, the new peer informs other peers incrementally. Conversely, when other peers' queries arrive at the new node, it receives information about the rest of the network. Thus, the system gradually evolves to a state which is the result of the global query workload. The dynamic evolution of the PIs is an open issue left as future work.

Based on the premise that each node chooses a set of keywords that is characteristic of its data, one can argue that this set will remain fairly stable over time. Periodically or when triggered by update events, a node may decide to inform its peers about new or modified local data. It constructs an update message and propagates this throughout the system along the edges of the join graph. When a peer receives an update, it processes it, replacing old entries in the PI or removing obsolete ones.

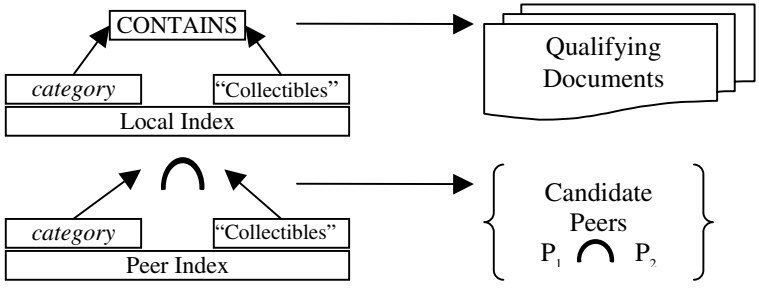


Fig. 3. Peer selection example for the query *//category*[*text*() *contains "Collectibles"*]*"*

4 Efficient Searches Using PIs

Using the peer indices constructed during node joining, nodes can direct searches to the nodes that are most likely to have results. This section describes how the system uses those indices. The simple example shown in Fig. 3 illustrates the basic mechanism of selecting candidate peers for a given query. To retrieve all XML documents that contain a category element that contains the word "Collectibles", one would use the query Q = *//category*[*text*() *contains "Collectibles"*]. Executing Q on the local inverted index, the system retrieves a collection of local documents that satisfy Q. Querying the PI yields two sets of peers, P_1 and P_2, for the element *category* and the word "Collectibles," respectively. Each peer in P_1 has *category* elements in its index and each peer in P_2 has "Collectibles" words in its index. The set $P_1 \cap P_2$ contains peers that are likely to have results, while it is reasonably certain, assuming all updates have been received, that peers in $(P_1 - P_2) \cup (P_2 - P_1)$ will not have any qualifying documents. Thus, if the system sends the query to peers in $P_1 \cap P_2$, we will get all possible results and at the same time will not take up resources of peers in $(P_1 - P_2) \cup (P_2 - P_1)$. Note that if either of the inputs of the intersection does not have any peers associated with it, the system reverts to a union, expanding the set of candidate peers in hope of finding results. For example, if P_2 were empty, the system would send the query to all or some of the *category* peers in case they return any results. Due to space limitations and the fact that our experimental data does not benefit from structural information in the PIs we do not present the structural details in this paper.

The semantic meaning of the tag *category* and the word "Collectibles" may not be uniform across the nodes that will receive the query. Hence, the results may contain

documents that will not match what the user had in mind but will definitely match the query correctly. A post-processing step on the results based on *themes* as proposed in [2] can achieve semantic correctness. In any case, the goal of contacting only a small subset of the nodes in the system will have been achieved. Section 5 shows how important it is to reduce the number of nodes contacted given a query.

5 Experimental Setup

We have compared several strategies for P2P query propagation on a 100-node cluster of PCs, which was used to stage 100-node and 200-node peer-to-peer networks. We create a scenario where a large number of data providers publish frequently changing data and desire to make it searchable by their peers. Four aspects make up our methodology: the scenario, the data, the peer-to-peer system set-up, and the simulation of users submitting queries to nodes in the system.

5.1 Scenario

Our scenario considers a large distributed auctioning system. Nodes are independent auction sites with subscribers who sell their items and bid on ongoing auctions. Each auction site specializes in specific categories of items. Users at each site would like to be able to pose queries that will be answered by all peers in the peer-to-peer system that may have relevant data. Sites may join the system by contacting any peer they choose. The result is a system that has a large number of nodes in a logical network similar to those in existing peer-to-peer systems.

```
<Item id="1045782232">
        <Name>US Navy Flag Cards</Name>
        <Category>Collectibles</Category>
        <Category>Cards</Category>
        <Currently>$24.00</Currently>
        <Quantity>1</Quantity>
        <Number_of_Bids>0</Number_of_Bids>
```

Fig. 4. XML document excerpt

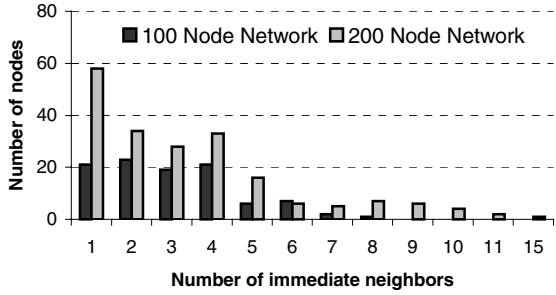
5.2 Data

Since we wanted to evaluate these strategies on real and not synthetic data, we crawled eBay's US site ([7]) and downloaded data for 6,500,000 items on auction at that time. This data was then used to feed a modified version of the XMark [20] benchmark data generator to augment the synthetic data it creates. This process produced about 45GB of XML data. Fig. 4 shows an excerpt of the item data.

We use the category hierarchy of eBay to divide the data among the peers. The category spectrum is divided into chunks that contain approximately the same number of items. Sets of chunks are assigned to each peer, which in turn, creates local inverted indices on elements and text words. We believe that this set-up represents a realistic environment of independent, interconnected, specialized auction sites.

Table 1. Linux cluster properties

System Property	Value
CPU	PIII 1GHz Dual Processor
Physical Nodes	100
Number of CPUs	200
RAM	1GB
Disk	IDE
OS	Linux 6.2 Kernel 2.2.19
Interconnection Network	100 Base-T
Java VM	IBM VM 1.3.0



	100 Nodes	200 Nodes
Mean fan-out	3	3.5
Diameter	9	10

Fig. 5. Distribution of join edges in the 100 and 200-node networks

Table 2. Experimental system parameters

Parameter	Value
Unique Element Tags	41
Unique Category Words	4081
Total Categories	5828
Categories Per Node	109 (± 65)
Category Words Per Node	128(± 81)
Documents Per Node	60,000-70,000
Keywords Per Node	~110,000
Average Local Index Size	~450 MB
Search Engine Buffer Pool Size	100 MB
Number of Users Per Node	12
User Think Time	15 seconds
User Type Time	9 seconds
Query Processing Threads Per Node	60
Result Collector Timeout	3 Minutes

Table 3. Correspondence of horizon size to query processing strategy

Fixed Radius Horizon Size	Query Processing Strategy
1	Routing indices (*H1*)
2-7	Intermediate levels of system information stored on each node (*H2-H7*)
Infinite	Full Summary Information (*FSI*)

5.3 System Set-Up

A 100-node dual-CPU cluster of PCs running Linux 6.2 (see Table 1) was used to emulate the nodes of a peer-to-peer system. For each peer-to-peer network, we create a join graph. Fig. 5 shows the basic characteristics of the networks used. Before running queries, nodes join with each other in order to produce the pre-defined network. While running, the nodes simulate wide area network delays (60ms exponentially distributed). A list of system parameters is shown in Table 2. Our prototype is implemented in Java with BerkeleyDB [1] used as the storage manager.

The experiments used *fixed radius horizon*s. This allowed the study of various levels of peer information on each node and their impact on overall system performance.
Definition 4: A node N has a *fixed-radius horizon* of h if it stores peer IDs only for nodes N_i for which the distance $d(N, N_i) \leq h$. The value h is the radius of the horizon.
With a fixed-radius horizon, a peer only stores information about peers nearby in the join graph and maps the remaining peers to nodes on the horizon boundary. The correspondence of fixed-radius horizon size to query propagation strategy is shown in Table 3.

5.4 Queries and User Simulation

Users in our scenario submit search queries in order to retrieve relevant documents from the system. Those queries can be arbitrarily complex, but contain predicates that allow the selection of candidate peers. Because nodes in our scenario specialize in auction item categories, we use the *category* element content to guide the selection of peers that receive queries. For example, the following query asks for all open auctions that contain an item named "Silver Dollar" that belongs to the "Collectibles" category: *//open_auction//item*[*category.text*() *contains "Collectibles"* AND *name.text*() *contains "Silver Dollar"*]. The peer selection predicate in this case is: *category.text*() *contains "Collectibles"*. For the experiments presented in this paper, queries with only category selection predicates were used. To generate the peer selection predicates for our queries, conjunctions of category words were used to generate a large pool of queries (about 200,000). Each node is assigned a number of queries and a number of users that randomly submit queries from the node's pool. The users work in cycles thinking and typing as described in [22] before submitting a query. Each user waits a specific time T after submitting a query for the network to return results. T is proportional to the moving average M of the response times of the last 100 queries. We set $T = 15 \times M$. This feedback mechanism allows the query rate to converge to a value the system can handle and models user behavior. Users adapt their waiting time to the system's performance.

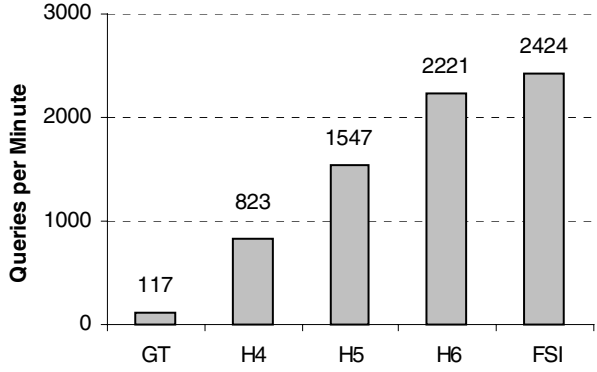


Fig. 6. 100 Nodes, throughput

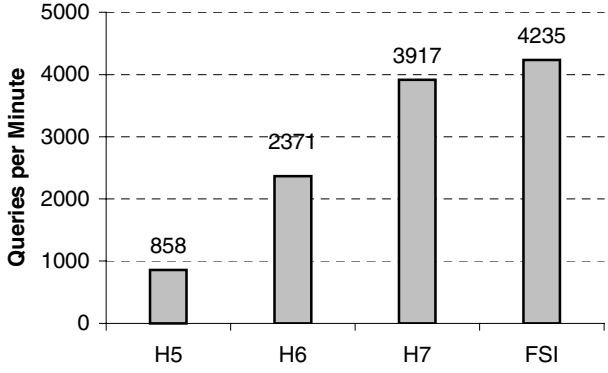


Fig. 7. 200 Nodes, throughput

6 Experiments

The experiments evaluate system performance for two network sizes (100 and 200 nodes) with various fixed radius horizon sizes and full summary information. Results from the naïve case of sending each query to every immediate neighbor (Gnutella-like) are also presented as the baseline approach. Before running experiments, a simulator was used to calculate the expected number of messages that peers would exchange during query processing. The inputs to the simulator are the indices created during joining. The results of the simulation were used to help verify the system experiments.

To take measurements, we used a time window *W*, usually 20-30 minutes long, during which all users in the system were submitting queries. For each configuration, the average query response time over all the queries within the time window *W* was computed along with the overall average query throughput. Query response time was defined to be the time to the first result. The average number of messages per query was also computed by normalizing the number of actual messages exchanged by the

total number of queries executed during the time window. All 95% confidence intervals of the measurements are within at most ±7% of the corresponding mean values.

The data for the 200-node network was obtained by cloning the data and indices from the 100-node network instead of redistributing and re-indexing. This approach allows us to alter only the size of the network. The node selectivity of queries, the size of the local inverted indices, and the number of keywords that nodes exchange remain constant.

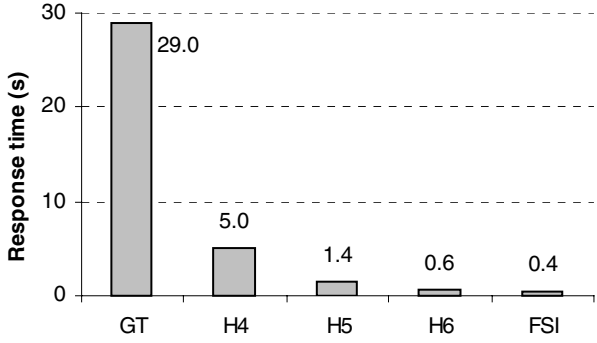


Fig. 8. 100 nodes, response time

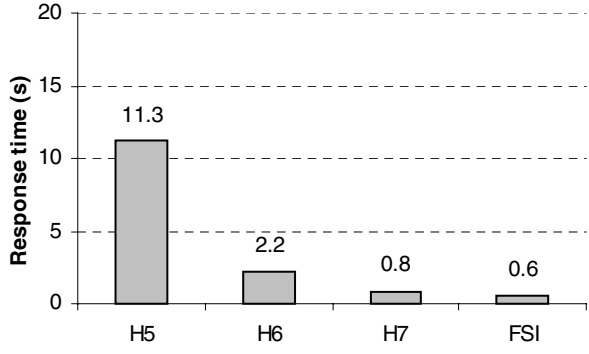


Fig. 9. 200 nodes, response time

6.1 Throughput and Response Time

We performed experiments with a number of different configurations, including the full summary index case (*FSI*), configurations with fixed width horizon sizes of 4 (*H4*), 5 (*H5*), 6 (*H6*), and 7 (*H7*), and a Gnutella-like configuration (*GT*) without any PIs. The results obtained are presented in Figs. 6, 7, 8, and 9. These results clearly indicate that the best performance is obtained by maintaining full summary information about each node in the system: In the 100-Node network the average throughput and query response times are 2,420 queries/min and 0.4 seconds, respectively. In the

200-Node network the corresponding values are 4,240 queries/min and 0.6 seconds respectively. The improvement in query throughput in the 100-node case is 2,071% over the Gnutella-like configuration *GT*. The response time is 72 times better. Note also that query messages in *GT* travel only for 5 hops in the network and thus retrieve only 55% of the results found by the *FSI* configuration. To retrieve all possible results, messages in *GT* would need to travel for even more hops, which would make the performance of this approach even worse. We also observed degrading performance as the radius of the horizon decreased, which is a result of the total number of messages processed by the system as discussed below. Note that *GT* is not reported for 200 nodes since the result of the queries were useless due to system overload.

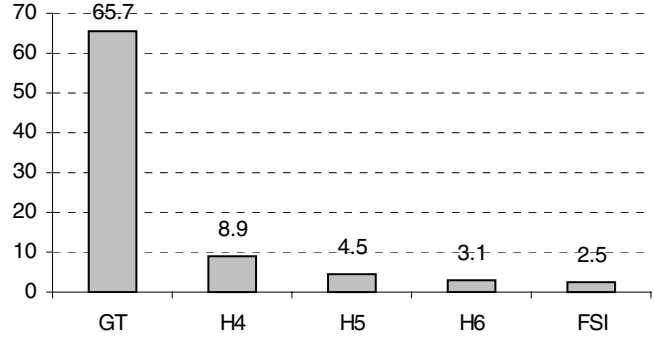


Fig. 10. 100 nodes, average number of messages per query

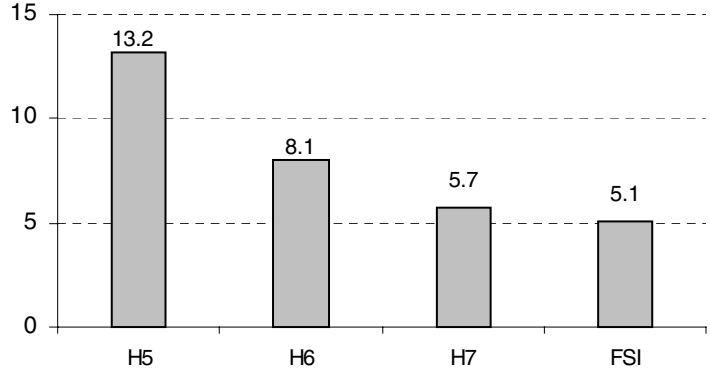


Fig. 11. 200 nodes, average number of messages per query

6.2 Analysis

The average number of messages per query that the system processed during the time window while measurements were being collected is presented in Figs. 10 and 11. The observed message counts fairly closely match the results obtained from the

simulation. As expected, the overall performance of the system and the number of query messages exchanged are strongly correlated with one another. The significantly large number of messages the system processes in the case of *GT* explains its unacceptable performance. In our experimental set-up, smaller horizons lead to as many as four times more messages (e.g. *H4* vs. *FSI* on 100 nodes).

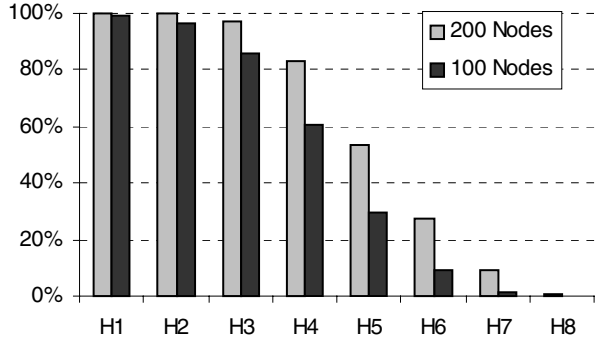


Fig. 12. Percentage of queries out of horizon

To explain the observed number of messages, we calculated the percentage of queries that left their origin horizon, shown in Fig. 12. These results suggest that the increase in the number of messages in smaller horizons is a consequence of the fact that more queries need to exit their respective horizons. Each time a query goes beyond the horizon of its originating node it incurs at least one extra message because it must be relayed to at least one node before reaching the destination. In our experiments, the performance of *H6* with 100 nodes and *H7* with 200 nodes is very close to *FSI* because only a small percentage of the queries leave the horizons of their originating nodes. Thus, the goal should be a system in which at least 90% of the queries stay within their origin horizon. Only *FSI* can guarantee this. If, however, *FSI* is not feasible due to large PI sizes, the system should evolve in a way that decreases the probability of a query going out of a horizon.

Table 4. Average number of messages per query

	GT	*H1*	*H3*	*H4*
100 Nodes	123	30	16	9
200 Nodes	279	110	66	31

Our experiments demonstrate significant performance increases when the number of messages passed at query time is reduced. While it is expected that a naïve query propagation strategy would be inefficient, it is surprising how poorly any strategy that routes queries through other peers performs compared to the *FSI*. Recent efforts ([5]) have focused on efficiently routing queries to the best neighbor. In our scenario, routing, which is essentially *H1*, performs much better than *GT* as can be seen in Table 4. We, however, feel that its use is not justified for two reasons: a) It attempts to replicate functionality already found in current Internet protocols and b) the number of messages it incurs is much higher than that using other small horizons as Table 4

demonstrates. *H1* is significantly better than *GT*, but at the same time much worse than *H3* and *H4*. Additionally, our experimental results demonstrate that *H4* does not perform as well as *FSI*. In order to achieve satisfactory query performance, we believe it is worth allocating the necessary resources needed for *FSI*.

6.3 Join Processing Times

Even though best performance during query processing is our main focus, the cost of joining must be considered. Consider the case where one additional node joins a network of 199 nodes. In our set-up, the new node spends about 4 seconds processing a join response from an existing node, while each existing node spends 0.3 seconds processing join updates. The new node will know summary information about all other nodes in the system in 4 seconds. All existing nodes propagate join updates prior to updating their index. Thus, all the nodes in the system will become aware of the new node after about $D \cdot d + 0.3$ s = 0.9 s (Network diameter D = 10, average network delay d = .06 seconds). As another example, assume that two existing networks of 100 nodes each join to form a network of size 200. In this scenario, each node will spend about 2s processing a join update message. The update message will contain peer information for 100 nodes. Hence, every node will have summary information about all other nodes in the new network after about 2.6s (D = 9, d = .06). It is clear that in an environment in which nodes do not exhibit the volatile behavior of users sharing music files performing the join we propose is essential for achieving better query throughput.

7 Related Work

Current research efforts with peer-to-peer systems can be largely attributed to the widespread use of file-sharing systems such as Napster [15] and Gnutella [8]. Napster is used for searching collections of media files using centralized indices to which clients connect and upload their file lists and search other users' lists. However, Napster is a hybrid peer-to-peer solution since it employs a centralized index for searches. OpenNap servers [17] work as Napster index servers, but forward queries they cannot answer themselves to a neighbor. A comprehensive study of an OpenNap network appears in [27]. They propose a mathematical model, which they validate on a network of 5 servers. Gnutella is a pure peer-to-peer application that allows searches to flow through a network of interconnected peers. In Gnutella, each peer forwards a search to all its immediate neighbors in a breadth first manner. This simple way of executing searches has been widely criticized for its abuse of network bandwidth. Our study quantifies how wasteful the flooding strategy can be. Recently, support for super-peers has been added to the Gnutella protocol. A study of super-peer networks appears in [28]. In both cases no query content based node selection is performed.

The attempt to reduce wasted bandwidth in the Gnutella network has prompted several research efforts. A simulation study of routing indices is presented in [5]. The goal is to choose the best neighbor of a node to forward the search until the desired number of results is reached. This approach only evaluates routing and does not explore directly contacting relevant peers. Techniques for improving Gnutella performance are presented [26]. Among them is a technique that indexes content of nodes in

the neighborhood. The index is not used for routing but for answering queries on behalf of other nodes.

Distributed lookup services have been investigated in [6], [13], [19] and [21]. Supported queries are restricted to object lookups by their keys. In addition, identifiers help route lookups to the relevant data pools. In contrast, our focus is on complex queries on data content. Distributed database systems ([18], [24]) have addressed query processing in a wide area environment, but usually assume full control over all the nodes in the system and the existence of detailed catalogs, and thus do not allow ad-hoc formation of peer-to-peer networks.

Sun's JXTA Search [23] provides searches of data sources that actively produce data, such as news sites. The goal is to retrieve the most up-to-date data, which is not possible using a fully centralized index like Google. At the same time, they try to avoid flooding those data sources with queries, by building indices on the queries a data source can answer. The indices reside on index servers (called hubs), to which affiliated data sources connect. There is no strategy for forwarding queries to other hubs, as they anticipate that nodes of similar topics will connect to the same hubs and that a small number of hubs is sufficient to index large numbers of information providers. However, they have not evaluated their approach on a large number of hubs.

8 Conclusions and Future Work

We have presented a performance study of a peer-to-peer system of autonomous XML search engines with a variety of strategies for P2P query processing. In essence, the join mechanisms pre-compute query destinations and store them in the peer indices. The intended use of our research prototype is as a low-level layer for our distributed query processing system that needs to discover all relevant XML documents for a given query in the philosophy of the Niagara System. Results from large-scale experiments on a pure peer-to-peer system demonstrate significantly improved query throughput and response time over current peer-to-peer architectures. The prototype we have implemented will be used as a test bed for future research into join algorithms, as well as various query processing techniques.

References

[1] BerkeleyDB. http://www.sleepycat.com.
[2] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. VLDB Journal 10(1): 48-71 (2001)
[3] J. Callan, M. Connell, A. Du. Automatic Discovery of Language Models for Text Databases. SIGMOD 1999 Conference.
[4] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD 2000 Conference.
[5] A. Crespo, H. Garcia-Molina. Routing Indices For Peer-to-Peer Systems. Tech Report http://dbpubs.stanford.edu:8090/pub/2001-48
[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. Wide-area cooperative storage with CFS. SOSP 2001
[7] eBay. http://www.ebay.com
[8] Gnutella Resources. http://gnutella.wego.com/.

[9] GoXML. http://www.goxml.com
[10] Google. http://www.google.com
[11] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, D. Suciu. What Can Databases Do for Peer-to-Peer. WebDB Workshop 2001.
[12] P.G. Ipeirotis, L. Gravano, M. Sahami. Probe, Count, and Classify: Categorizing Hidden-Web Databases. SIGMOD 2001 Conference.
[13] J. Kubiatowicz et al. OceanStore: An Architecture for Global-Scale Persistent Storage. In Proc. ASPLOS 2000.
[14] J.F. Naughton, D.J. DeWitt et al. The Niagara Internet Query System. IEEE Data Engineering Bulletin 24(2): 27–33(2001)
[15] Napster. http://www.napster.com
[16] B. Nguyen, S. Abiteboul, G. Cobena, M. Preda. Monitoring XML Data on the Web. SIGMOD 2001 Conference
[17] OpenNap Project. http://opennap.sourceforge.net
[18] M. T. Özsu, P. Valduriez. Principles of Distributed Database Systems, Second Edition. Prentice-Hall 1999
[19] A. Rowstron, P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. SOSP 2001
[20] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.
[21] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proc. SIGCOMM 2001.
[22] TPC-C Benchmark Standard Specification Revision 5.0.
[23] S. Waterhouse. JXTA Search: Distributed Search for Distributed Networks. White Paper http://search.jxta.org
[24] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Linsey, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, R. Yost. R*: An Overview of the Architecture. IBM Research Report RJ3325.
[25] XML Path Language (XPath) 2.0 http://www.w3.org/TR/xpath20/
[26] B. Yang, H. Garcia-Molina. Efficient Search in peer-to-peer networks. In Proc. ICDCS 2002.
[27] B. Yang, H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In Proc. VLDB 2001.
[28] B. Yang, H. Garcia-Molina. Designing a Super-Peer Network, In Proc. ICDE 2003.
[29] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001 Conference.