The InfoMatrix: Distributed Indexing in a P2P Environment

Ref. No: 154

ABSTRACT

We study the problem of enabling efficient queries over dynamic data distributed over a wide-area peer-to-peer network. We show that existing solutions to this problem impose either high querying cost, or high index-maintenance cost. We introduce new indexing structures that allow dynamic trade-offs between the cost of maintaining the index in the face of data dynamism, and the querying benefit obtained from the index. Furthermore, we show how these indexing structures are optimized for the underlying physical network, so as to reduce bandwidth usage and minimize the latency of queries. We use experiments on real workloads to demonstrate that our indexing structures improve on the performance of existing solutions by an order of magnitude.

1. INTRODUCTION

A peer-to-peer(P2P) system consists of a large, dynamic set of computers, called *nodes*, spread over a wide-area network. The scale and dynamism of the system precludes a node communicating directly with all other nodes. Instead, nodes are interconnected in an *overlay network* with each node allowed to communicate directly only with its neighbors on the overlay network. Every node "owns" some set of tuples conforming to a fixed relation schema, thus implicitly defining a horizontally partitioned relation R. This paper discusses an architecture called the *InfoMatrix* which uses distributed indexes to support efficient queries over R.

To illustrate the InfoMatrix, let us consider the most popular P2P application: file sharing. A prototypical file-sharing system has millions of participant nodes spread all across the internet. Nodes may be highly dynamic with a median lifetime as low as one hour. Each node "owns" hundreds of files, with each file being associated with a set of keywords. Thus, each file may be represented as a set of tuples $\langle nodeId, fileId, keyword \rangle$, one tuple for each keyword associated with the file.

Any node may issue a query consisting of one or more keywords and, in response, expects to receive a list of references to files in the system that match all these keywords. However, the node may not require *all* matches to the query, but only a specified number of query results, say k. Such queries are called *partial-lookup selection queries*.

There have been two different approaches proposed for answering such queries. In the Gnutella-style [1] approach, a node n initiating a query q simply sends out q to all other nodes (or some fraction of the nodes) in the system. Each node then evaluates q on its own data and sends the results to node n. This approach suffers from an extremely high query cost, since each query may need to be processed by a large number of nodes in order to find a sufficient number of results for the query.

The second approach is to use a globally distributed index. Imagine building a global inverted index over all keywords, and partitioning this index, by keywords, across the nodes in the system. For example, node n_a stores the inverted list for the word "apple", node n_b for the word "boy", and so on. When node n has a query q, it simply needs to contact the nodes maintaining the inverted lists for the keywords in q. Queries are very efficient, since very few nodes need to be contacted to answer a query. However, as we will show, this approach suffers from a high index-maintenance cost. As content in the system changes continuously, due to nodes joining and leaving, the index needs updating to ensure that queries do not return "stale" answers that no longer exist in the system, and can return answers that are newly available in the system. We will see that this index-maintenance cost is often orders of magnitude higher than the query cost, and does not scale well with the number of tuples owned by each node.

Our objective is to devise a system that can dynamically trade off index-maintenance cost against the benefits obtained for queries, in order to minimize the *total cost* of index maintenance and query execution. In addition, we require that the scheme scales well with the *number of tuples owned by each node*, since we expect the data stored per node to grow rapidly over time, even more so than the number of participant nodes in the system.

We now illustrate the basic intuition behind our solution, the InfoMatrix, which offers the above desiderata. At its heart, the InfoMatrix can be viewed as a two-tier structure as shown in Figure 1. Nodes are broken up into independent *domains*, and nodes within a domain build a distributed index *over the content stored in that domain*. A query is first evaluated within the domain it is posed in, using the efficient index structure available in the domain. If insufficient

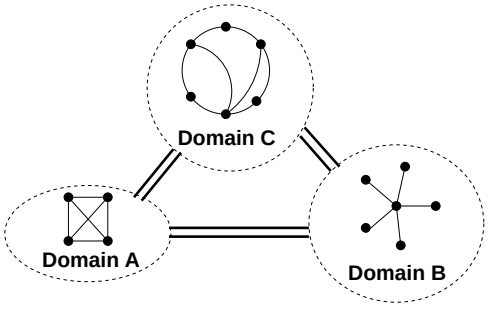

Figure 1: The Basic InfoMatrix Structure

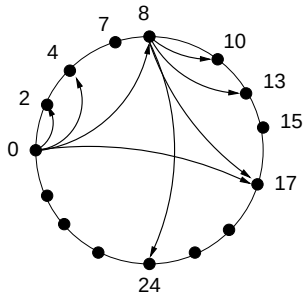

Figure 2: An Example Chord Network

answers are found, the query is then efficiently forwarded to more and more domains, and efficiently evaluated using the intra-domain index within each of them. The query ends either when a sufficient number of answers is found, or when all domains have received the query.

Note that the inter-connection of domains shown in the figure is only conceptual, and is actually realized as part of the overlay network interconnecting nodes, while ensuring that each node has only a small (logarithmic) number of links to other nodes. We show that this basic architecture, in combination with a new index-maintenance mechanism, a smart query-propagation algorithm, and the right choice of domain size, can provide up to an order of magnitude improvement in terms of overall system overhead, in comparison to a globally distributed index. In addition, the system scales gracefully as the amount of data per node increases, provides query answers with lower staleness, and ensures a more uniform distribution of index-maintenance overhead across nodes.

However, this simple architecture is not the end of the story. The InfoMatrix improves on this basic structure in order to provide the following crucial properties:

- Query and index-maintenance traffic is optimized for a heterogeneous physical network, where the latency, and the bandwidth cost, of communication between different pairs of nodes may be different.
- The architecture offers *nearest-k* semantics, where a node n issuing a query desires the k answers closest to it on the physical network.
- The number of domains used can vary dynamically to achieve the right trade-off between index-maintenance and query costs.

In addition, complex partial-lookup queries that require index intersection, e.g., multiple-keyword queries, are more efficient on the InfoMatrix than in global index structures. The InfoMatrix also supports applications that require indexing constrained by trust relationships, access control, or privacy. We discuss some of these applications in Section 6.

The rest of this paper is organized in an "onion" structure as follows. In Section 2, we introduce prior work on distributed indexes and structured networks that we build on in our solution, as well as other related work. In Section 3, we present a first-cut design of the InfoMatrix, assuming a fixed set of domains, simple selection queries with a single equality predicate, and a homogeneous physical network. Section 4 explains how this design is improved to deal with a heterogeneous physical network by the incorporation of multi-level hierarchies. Section 5 extends the design further to deal with dynamic definitions of domains. Section 6 describes further extensions to the InfoMatrix, and discusses a variety of applications in which the InfoMatrix can prove useful. We present an experimental evaluation of the InfoMatrix on real workloads in each of Sections 3, 4 and 5.

2. BACKGROUND AND RELATED WORK

There has been a lot of recent work on building structured overlay networks and Distributed Hash Tables (DHTs) that we will build on in this work. We begin by explaining one such system, called Chord [19].

Chord: Chord [19] is a structured overlay network defined as follows. Say there are n nodes in the system. Each node is assigned a unique *ID* drawn at random from a circular N-bit identifier space $[0, 2^N)$. (Identifiers on the circle are imagined as being arranged in increasing order clockwise.) Figure 2 depicts an example of nodes in a 5-bit identifier space. The *distance* from a node m to a node m' is the clockwise distance on the circle from m's ID to m''s. Each node m maintains a link to the closest node m' that is at least distance 2^i away, for each $0 \leq i < N$. For example, node 0 establishes a link to the closest nodes that are a clockwise distance $1, 2, 4, 8$ and 16 away, resulting in links to nodes $2, 4, 8$ and 17. Similarly, node 8 establishes links with nodes $10, 13, 17$ and 24.

This Chord structure allows efficient routing of a message from any node to any other node by simple greedy, clockwise routing. For example, when node 0 desires to reach node 15, it would use greedy routing to send the message to node 8, which would forward it to node 13, which would then be sent along to node 15. This mechanism requires only $O(\log n)$ messages for routing between any arbitrary pair of nodes. Moreover, when a node joins or leaves the system, it requires only $O(\log n)$ messages to restructure the overlay network appropriately for the new set of nodes.

Chord was originally proposed as a means of maintaining a distributed hash table. Data keys are hashed into the N-bit identifier space, and we let each node store data falling in the hash bucket ranging from its ID to the next higher ID. For example, node 0 would store data in the bucket $[0, 2)$, node 2 would store data in $[2, 4)$, and so on. A query for a specific key can then be routed to the node responsible for the key's bucket, thus allowing efficient key lookup.

Broadcast on Chord: Although Chord was proposed for routing, an obvious extension also allows efficient *broadcast* of messages from one node to all other nodes. (This extension is described in the context of Pastry in [11].) A node desiring to broadcast a message sends the message to all its neighbors, and requires each neighbor to recursively broadcast the message to a smaller fragment of the ring.

For example, node 0 broadcasts a message by sending it to all its neighbors: nodes 2, 4, 8 and 17. Node 2 would be responsible for sending the message to nodes in the range $[2, 4)$, node 4 for the range $[4, 8)$, node 8 for the range $[8, 17)$ and node 17 for the range $[17, 32)$. Each of these nodes recursively forwards the message to all its neighbors falling within its specified range. For example, node 8, being responsible for the range $[8, 17)$, would send the message to nodes 10 and 13, appointing them responsible for the ranges $[10, 13)$ and $[13, 17)$ respectively.

It is easy to show that the above mechanism results in each node receiving the message exactly once. Moreover, if each node transmits one message per time unit, every node receives the broadcast message within $O(\log n)$ time units.

2.1 Related Work

Distributed Hash Tables [12, 19, 15, 7], such as Chord, have been proposed as a substrate for a variety of distributed applications, including archival storage [16], distributed file systems [4], cooperative caching [9], multicast [11, 14] and publish-subscribe systems [17].

The PIER project [8] uses DHTs specifically for the problem of building distributed indexes over relational data to enable single and multi-table queries. References [20] and [21] propose building Information Retrieval systems on top of DHT structures to enable complex IR queries. All these works are complementary to ours in that they enable more complex queries on top of basic index structures, and do not consider the question of index maintenance cost due to data dynamism. We believe that many of these works can be adapted to use the InfoMatrix instead of standard DHT implementations in order to make the indexes efficient when data is dynamic. Other work on P2P databases tackles the data integration problem among heterogeneous databases [6]. These issues are orthogonal to the question of efficient indexing investigated in our work.

File-sharing systems originated with Napster, which used a central index of all content. Gnutella [1] and its variants chose the other extreme, with each node simply indexing its own content, and queries being flooded across nodes. KaZaa [2] extended the Gnutella approach with the use of *supernodes* which serve as a proxy for less stable nodes with lower bandwidth. (Our InfoMatrix architecture also extends directly to allow solutions that distribute the index only over such supernodes, rather than relying on all nodes in the system.) Both Gnutella and KaZaa suffer from high query-execution cost and poor query recall. Overnet [3] is a file-sharing system built on DHTs. Our experiments on Overnet suggest that it has low query precision due to the number of stale answers, but has excellent recall for single-keyword queries.

Hybrid solutions such as YAPPERS [5] have been proposed to introduce a middle ground between DHTs and Gnutella, and enable efficient partial-lookup queries while reducing index-update costs. Our solution extends the design philosophy of YAPPERS, but improves on it by designing much larger, non-overlapping indexes, while providing stronger guarantees on query and update costs.

3. THE INFOMATRIX: THE FIRST CUT

We now describe a highly simplified version of the InfoMatrix to explain its fundamental architecture. We assume that there is a fixed number of domains k, each with a unique *domain name*. (For example, the domain name could simply be an integer between 1 and k.) Each node is assigned to one of these k domains when it joins the system, for example, at random. Assume, further, that the physical network is homogeneous and that the cost of communication between any pair of nodes is identical. Finally, assume that we desire to support only simple selection queries with an equality predicate on a specific attribute.

Our objective is to build a distributed index among the set of nodes in each domain, and devise a querying mechanism that efficiently executes queries to find some, or all, answers available in the system. The architecture can be broken up into two parts: the intra-domain structure and the inter-domain structure. We now describe each of them.

3.1 The Intra-domain Structure

The nodes within each domain organize themselves in a Chord structure, just as described in Section 2, in order to form a distributed hash index over all the content in that domain. Thus, each node in a domain has a unique intra-domain ID, and stores index entries for all tuples within the domain whose index attribute hashes to a value between its ID and the next larger ID. This intra-domain structure enables any node s to initiate a query q and receive all the answers for q available within the domain.

For example, if node s initiates a query for a keyword "apple", it first hashes the keyword and then uses Chord to route the query to the node a responsible for the hash bucket containing the keyword. Node a then responds to s with the list of tuples matching the keyword. We denote this entire operation by $\mathit{FindIntraDomainAnswers}(s, q)$.

The intra-domain Chord structure adapts itself as nodes join and leave the network. (A joining node needs to know one other existing node in the domain. We discuss this bootstrap problem in Section 3.3.) When a new node joins a domain, or an existing node leaves, two additional operations need to take place: (a) the entries in the index structure may need to be re-distributed across nodes, in order to allow the new node to maintain a portion of the index, or to make up for the loss of the existing node, and (b) the index itself needs updating to ensure that the new node's *content* is indexed, or that the old node's *content* is removed from the index. Problem (a) has been studied in past literature [19] and our focus in this work is on problem (b).

Let us examine how the index entries are updated with node dynamism. When a new node m joins the domain, updating the index is straightforward. For each tuple that m owns, it hashes the key attribute of the tuple, and sends the index entry for the tuple to the appropriate node by routing it using the Chord structure. Updating the index when a node leaves is more interesting. We now describe two approaches to this problem.

3.1.1 The Time-Out Mechanism

The traditional approach to index updates is the time-out mechanism [8]. Each index entry inserted into the system has a time period, say T minutes, for which it stays "alive". At the end of this period, the index entry "times out" and is removed by the node storing the entry. Nodes that are alive for long periods of time will "refresh" the index entries for their content every T minutes. Nodes that leave the system will not refresh their index entries and, thus, the entries corresponding to their data will be removed within

T minutes of their leaving the system.

The size of the time period T offers a trade-off between the cost of maintaining the index and query precision. If T is small, the maintenance cost is high since nodes have to refresh index entries frequently, but there are very few false positives. If T is large, the maintenance cost is low, but there may be more false positives from stale index entries.

The index-maintenance cost for a node in this scheme is measured as the number of refresh messages sent by it per second. (We assume that a node can directly send refresh messages to the appropriate destinations, instead of having to route these messages through the Chord network; so, refresh messages do not contribute to additional routing traffic on the Chord network.) Note that this maintenance cost captures not only the bandwidth used in maintenance, but, more importantly, the operating-system overhead. To illustrate, the bandwidth necessary to transmit, say 1000, 64-byte messages per minute is a relatively small 8kbps. However, sending these 1000 messages to 1000 different destinations each minute, requires the setting up and tearing down of $1000/60 \approx 16$ TCP connections per second, which can lead to considerable operating-system overhead.

Observe that the maintenance cost for a particular node is directly proportional[1] to the number of tuples stored by that node. This exposes two problems: (a) the maintenance cost increases linearly with the number of tuples per node, and (b) a skewed distribution of tuples across nodes leads to a skewed distribution of maintenance costs across nodes.

3.1.2 The Update-Broadcast Mechanism

The relatively small number of nodes in an InfoMatrix domain, compared to that in a global index, offers an alternative approach for index maintenance that has not been studied in the P2P literature. Whenever a node n leaves the domain, the successor of n on the Chord ring (which also happens to be a neighbor of n) broadcasts the information about n's departure to all the nodes in the domain (using Chord broadcast, for example). Each node can then eliminate index entries corresponding to tuples of n. We call this the *update-broadcast* mechanism.

One may wonder how n's successor learns of the departure of n in the first place. This is achieved by the exchange of periodic "keep-alive" messages between adjacent nodes on the Chord ring, which is necessary even otherwise for the maintenance of the Chord network [19, 4, 8]. In this case, the periodicity of these keep-alive messages governs the delay in updating the index to eliminate stale entries and, consequently, the query precision. Typically, keep-alive messages are very frequent (we assume at least one a minute), and we will see that the query precision is consequently very close to 100%. (Note that exchanging keep-alive messages is very efficient in practice, since each node can maintain a permanent TCP connection with its adjacent node in Chord, and exchange messages on this link.)

The maintenance cost using update broadcast is a linear function of the number of nodes in a domain, and becomes higher than the cost of the time-out mechanism for large domains. We show the following theorem, quantifying the exact domain size up to which update-broadcast is more efficient, in terms of the number of messages per second, than the time-out mechanism. The proofs of all theorems presented here are in the extended version of the paper.

[1]This is true while the number of tuples per node is much smaller than the total number of nodes.

THEOREM 1. *For a domain of d nodes, with an average of k tuples per node, mean node lifetime T_l and time-out period T_o, the time-out mechanism is more efficient than update broadcast if and only if $(1 - 1/d)^k > 1 - T_o/T_l$.*

COROLLARY 1. *The time-out mechanism is more efficient than update broadcast only if $d > k \cdot T_l/T_o$.*

Update broadcast offers three advantages over the time-out mechanism. First, the maintenance cost of update broadcast is *completely independent of the number of tuples owned by each node*, thus allowing the system to scale up efficiently even as the number of tuples per node increases. Moreover, even when nodes have a skewed distribution of tuples, the maintenance cost for the different nodes still remains uniform. Second, update broadcast offers higher query precision than is possible with reasonable time-out values. Third, there is zero additional overhead for index update, when index entries are replicated across multiple nodes to improve query recall [19, 4].

In Section 3.4, we evaluate both the time-out and update-broadcast mechanisms, and show that the update-broadcast mechanism is more efficient for domain sizes up to 5000. However, update broadcast can be made more efficient than the time-out mechanism for even larger domain sizes, by aggregating information about multiple nodes leaving the domain in a single message; we exploit the fact that sending a single, large message is more efficient, in terms of CPU usage and bandwidth, than transmitting multiple, small messages. The details of these optimizations can be found in the extended version of this paper.

3.2 The Inter-domain Structure

So far, we have seen how any node may find all the answers to a query that are within its own domain. We now describe how nodes are interconnected to enable partial and total-lookup queries, allowing a node to find a desired number of answers, or all answers to the query in the entire system.

Our solution relies on iterative broadcast, and has a simple intuition. A node first finds all answers to the query within its own domain. If the number of answers proves insufficient, it attempts to find answers from (roughly) one additional domain (bringing the total number of domains searched to two). While the number of answers found proves insufficient, the node keeps doubling the number of domains it searches (in expectation) until either a sufficient number of answers are found, or all the domains have been searched. When a node desires a total lookup, the query is simply broadcast to all the domains. We first describe the interconnection structure necessary to achieve this broadcast, and then discuss the actual broadcast algorithm.

3.2.1 The Interconnection Structure

In order to support the iterative broadcast of queries, we will once again use the Chord structure, but this time as a *conceptual* interconnection network of *domains*. First, let us define a conceptual link going from one domain to another. Consider two domains D_1 and D_2, and let the set of nodes in the two domains be N_1 and N_2 respectively. A "link" from domain D_1 to domain D_2 implies that, for each node $n_1 \in N_1$, there exists some node $n_2 \in N_2$, such that node n_1 links to node n_2 on the overlay network. Thus, a conceptual

link from one domain to another is realized as a *set* of links going from nodes in the first domain to nodes in the second.

We can now define the inter-domain structure. The *domain name* of each domain is hashed using a pre-specified hash function to produce a *domain identifier*. Domains are then arranged in a circular space using their domain identifiers, and are interconnected in a Chord structure. Each Chord link from one domain to another is implemented by a set of inter-node links, as described above. The following theorem bounds the number of actual links established by each node (the node's *out-degree*) in this structure.

THEOREM 2. *If the total number of domains is k, the out-degree of each node is $O(\log k)$ with high probability.*

Optimizing conceptual links: As an optimization, we require a conceptual link from domain D_1 to D_2 to be implemented by a link going from each node n_1 in D_1 to the node n_2 in D_2 *with the closest intra-domain ID*. In other words, node n_1 would attempt to connect to the node in D_2 which manages approximately the same portion of the hash space as itself. This optimization ensures that, if node n_1 is responsible for answering a query q in D_1, node n_2 (or one of its neighbors in D_2) is likely to be responsible for answering q in D_2. Thus, q may be processed in D_2 using only a constant number of messages if it is despatched to node n_2.

Dealing with Node Dynamism: Our inter-domain structure may need to be adjusted as nodes join and leave the system. Such modification is fairly straightforward. The *conceptual* inter-domain structure remains the same, irrespective of the number of nodes in each domain[2]. As a new node joins, it simply needs to set up a link to one node in each appropriate domain, which is easily implemented just as in Chord. Similarly, when a node leaves a domain D, other nodes that link to it need to link to a different node in domain D instead. Again, this is easily achieved.

3.2.2 Iterative Broadcast

The broadcast of a query from one domain to all others is similar to the broadcast on Chord described earlier, but with two differences: (a) broadcast occurs on the conceptual inter-domain Chord structure rather than on a normal Chord network, (b) broadcast occurs iteratively, doubling the number of domains reached at each step.

Algorithm 1 describes the iterative broadcast algorithm, designed for partial-lookup queries. The starting node s first finds answers to the query q within its own domain. If the number of answers proves insufficient, it forwards q to its first neighbor domain D_1 (in the conceptual Chordnetwork), and requests this neighbor to propagate q along to all domains up to its next neighbor D_2 (using Algorithm 2). If the answers are still insufficient, it forwards the query to D_2, requiring D_2 to propagate q to all domains up to D_3, and so on, until either sufficient answers have been found, or all domains have been reached.

Note that, in step 7 of this algorithm, the source node s needs to "wait" till the query q has been propagated to, and processed by, the desired set of domains, before node s proceeds to the next step. This waiting can be implemented in multiple ways, and we omit details.

The following theorem bounds the amount of time taken, and the number of messages used, to propagate a query using this iterative broadcast algorithm.

[2]The special case when no node exists in a domain is also easily handled.

Algorithm 1 FindQueryAnswers(StartNode s, Query q)

1: Answers=*FindIntraDomainAnswers*(s, q)
2: Let the domain of s be D, and let n be the node responsible for q in D.
3: Let the neighbor domains of D be $D_1, D_2, ...D_i$, sorted in increasing order of distance from D. Let $D_{i+1} = D$.
4: Let the corresponding neighbors of n in these domains be $n_1, n_2, ...n_i$.
5: **for** $j = 1$ to i **do**
6: If sufficient answers, **break**.
7: Answers=Answers+PropagateQuery(n_j, D_{j+1}, q,s)
8: **end for**
9: return Answers;

Algorithm 2 PropagateQuery(CurrentNode n, LimitOfPropagation l, Query q, SourceNode s)

1: Let the domain of n be D_0, with the neighbor domains being $D_1, D_2, ...D_j$, sorted in increasing order of distance from D.
2: Let the corresponding neighbors of n be $n_1, n_2, ...n_j$.
3: Let $k \leq j$ be the largest value such that D_k is closer to D than the propagation limit l.
4: **if** $k > 0$ **then**
5: PropagateQuery(n_k, l, q, s)
6: **end if**
7: **for** $x = k - 1$ downto 1 **do**
8: PropagateQuery(n_x, D_{x+1}, q, s)
9: **end for**
10: *FindIntraDomainAnswers*(n, q) and return them to s

THEOREM 3. *If each node can forward one message per time unit, a query initiated by a node is propagated to p domains in $O(\log^2 p)$ time units with high probability. The propagation of the query to these p domains requires exactly $p - 1$ messages.*

Note that this inter-domain structure can also be used to perform non-iterative broadcast for *total-lookup* queries. This is achieved by two simple modifications to Algorithm 1: (a) node s does not wait to receive answers in step 7, and simply propagates the query to all its neighbors, (b) the neighbors are contacted in reverse order from n_i to n_1. Query propagation for total lookup may be achieved in $O(\log k)$ time units, where k is the number of domains. In partial-lookup queries, the fact that we do not know the number of domains to be reached in advance creates the additional overhead in terms of time.

3.3 The Overall Structure

Having seen the intra- and inter-domain interconnection structures and querying algorithms, we present the following theorem on the *overall structure* of the interconnection network, and *overall cost* of queries.

THEOREM 4. *If there are n nodes in the system distributed uniformly across k domains,*

(a) the total out-degree of each node is $O(\log n)$ with high probability (w.h.p.).

(b) a query initiated by any node takes $O(k + \log(n/k))$ messages w.h.p.

(c) a total-lookup query is answered in $O(\log n)$ time units w.h.p., while a partial-lookup query is answered in $O(\log^2 k + \log(n/k))$ time units w.h.p.

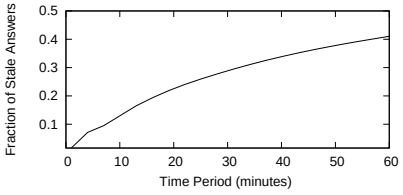

Figure 3: Fraction of stale results as a function of time-out period

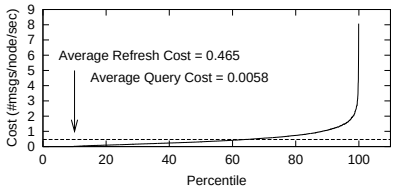

Figure 4: Distribution of refresh costs

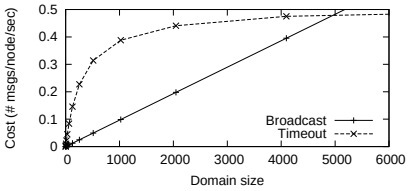

Figure 5: Refresh cost as a function of domain size

So far, we have not said exactly what the number of domains k *should be*, in order to achieve optimal performance. The best value of k for a given number of nodes n, is application-dependent. In some application scenarios, like the ones we describe in Section 6, k is predetermined by the application. In other applications, such as file-sharing, we will see that it is beneficial to dynamically vary k as n changes. Section 5 discusses how this may be achieved.

Bootstrap: One final question to be resolved is that of bootstrap: How does a new node joining the system locate a node in its own domain, in order to join the intra-domain Chord structure, and set up inter-domain links? This problem is solved using *routing on the conceptual inter-domain structure*. When a new node m joins, we assume that it (a) knows its own domain name, and (b) knows some existing node m' in the system. Node m then requires m' to route a message to m's domain identifier; this routing takes place on the conceptual inter-domain Chord structure, and helps in locating a node m'' in m's domain. Node m can then communicate with node m'' to set up its intra- and inter-domain links, just as in Chord [19].

3.4 Evaluation

We now evaluate of this first cut of the InfoMatrix, using real data gathered by Saroiu et al. [18] in a study of the Gnutella file-sharing network. The study provides information about a set of 3791 hosts (nodes) participating in the Gnutella network, together owning more than 400,000 files. The data includes the list of all files being shared by each node, and the lifetime of each node. Nodes have 107 files each on average, with each file name averaging 6.5 keywords. The elimination of stop words and one- and two-letter words from file names reduces the number of distinct keywords per node to about 300. The lifetime of nodes follows a skewed distribution, with the average lifetime being about 3 hours, and the median lifetime being 1 hour.

We extrapolate this data to simulate larger systems with n nodes, for arbitrary n, assuming that each node's lifetime characteristics, and set of files, follows the same distribution as that of the measured data. Many of our experiments will be on $n = 32768$ node systems, although our experiments on scalability vary n, going all the way up to 1 million nodes.

Our query workload is obtained from a study of Gnutella by Yang et al. [24] which gathered a trace of 100,000 queries being executed on the network . We obtain the query rate from a different study by Yang et al. [23] on the OpenNap system, which suggests that the rate is 0.00083 queries per node per second (one query per node per 20 minutes). We present results only on the single-keyword queries in the workload. We note that the InfoMatrix architecture is even more efficient than global indexes for multi-keyword queries, and therefore, our omission of such queries in this evaluation only understates the utility of the InfoMatrix.

3.4.1 Evaluating the Global-Index Approach

Our first undertaking is to evaluate the maintenance cost of the traditional global-index approach. With a global index, update broadcast is not feasible due to the high cost of broadcast, and we therefore evaluate it in conjunction with the time-out mechanism.

Time-out vs. Staleness: Our first experiment quantifies the trade-off offered by the time-out mechanism between query precision and the cost of periodic refresh messages, for different values of the time-out period. This trade-off is independent of the number of nodes in the system. Figure 3 depicts the fraction of stale answers returned by a query, as a function of the time-out period used to refresh index entries. We see that if the time-out period is one hour, more than 40% of the answers are stale (implying that precision is 60%). In order to achieve a staleness of under 10%, the time-out period needs to be smaller than 10 minutes.

Global-Index Costs: Next, we evaluate the global-index structure for a system of 32K nodes, using a time-out period of 10 minutes to obtain nearly 90% precision in query answers. We present the distribution of refresh costs, as well as the average query and refresh costs in Figure 4. The curve plots the refresh cost of a node against its percentile rank in terms of refresh cost. The right endpoint of the curve shows that the node with the highest refresh cost has to send about 8 refresh messages per second. In fact, this node needs to send refresh messages to about 5000 different nodes over a period of 10 minutes, which can lead to considerable overhead[3].

Even the average refresh cost is a factor of 90 higher than the average *query cost*, which is so small that it coincides with the x-axis in the figure. Moreover, as the system scales up to a point where each node has an average of 3000 keywords, we expect the average refresh cost to go up ten-fold. We conclude, therefore, that the use of a global index leads to a high, non-uniform overhead on the nodes, while the average refresh overhead is nearly two orders of magnitude higher than the query cost.

3.4.2 Evaluating the InfoMatrix

We now evaluate the index-maintenance cost, the query cost, and the overall system cost on the InfoMatrix for different domain sizes.

Index-Maintenance Cost: Figure 5 depicts the index-maintenance cost of the update-broadcast mechanism, as well as the time-out mechanism, for different domain sizes.

[3]An alternative would be to route these refresh messages through the Chord network, which increases the number of refresh messages processed per node by a factor of 7.5, but reduces the number of TCP connections necessary.

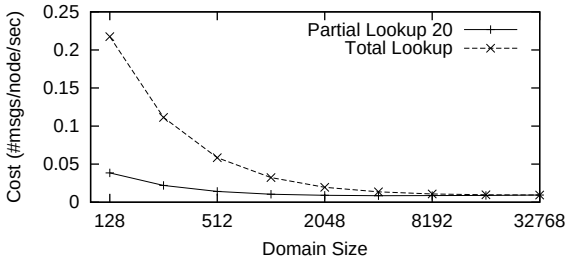

Figure 6: Cost of total and partial lookup queries as a function of domain size

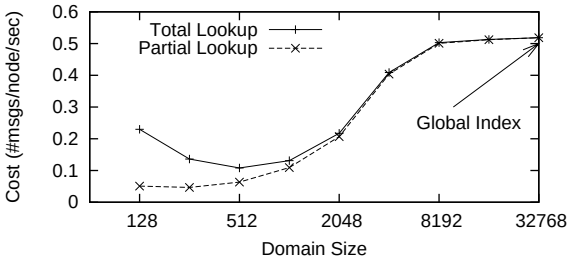

Figure 7: Overall cost as a function of domain size

(We assume that no optimizations are performed to reduce the cost of broadcast.) We see that the cost of the broadcast increases linearly with the size of the domain, and exceeds the cost of the time-out mechanism for domain sizes larger than 5000. This is in agreement with the estimate obtained from Theorem 1, plugging in the appropriate values for the relevant parameters.

Note that the precision of queries (not shown in figure) when using update-broadcast is more than 99%, assuming keep-alive messages are exchanged once a minute; in contrast, the time-out mechanism offers only a precision of about 90%. For domain sizes smaller than 1000, we see that update broadcast uses only one-fourth as many messages as the time-out mechanism *for the same domain size*. In comparison to the refresh cost in a global-index structure (Figure 4), we observe that the update cost for these domain sizes is an order of magnitude smaller. Moreover, this cost is independent of the amount of data stored at each node.

Query Costs: Figure 6 depicts the cost of queries, in terms of the number of messages processed per node per second, on a 32K-node network for different domain sizes. The figure depicts the cost of both total-lookup queries, which need to find all answers for a query, and *partial-lookup queries* which are terminated after finding the first 20 query answers. (Queries with fewer than 20 answers are equivalent to total-lookup queries.) Not surprisingly, we see that the cost of queries decreases as the domain size increases.

When the domain size is very small (128 nodes), the cost of total lookup is fairly high (0.22 messages/second). (Note, however, that even this high query cost is less than the update cost due to refresh messages in the time-out mechanism, as seen in Figure 4.) As the domain size increases, the cost of total lookup falls off drastically, and is less than 0.05 messages/second for domain sizes larger than 1000 nodes. Even more interestingly, the cost of partial lookup is extremely "flat", suggesting that there are so many answers available for most queries, that it is sufficient to use small domain sizes and query a small set of domains.

Overall Cost: Having seen that update costs increase with domain size while query costs decrease, we now show the *overall cost* of queries and updates in order to understand the total system overhead in maintaining the index and executing queries. We define the overall cost simply as the sum of the total query cost, the index-maintenance cost, and the cost of maintaining the interconnection structure as nodes join and leave the system, with all costs measured in terms of the number of messages per node per second.

Note that this calculation leaves out the cost of periodic keep-alive messages between adjacent nodes (recall that we assume a periodicity of one message per minute), which is the same for all domain sizes, and has very little cost, as discussed earlier. We also omit the costs of replicating index entries, which only strengthens the case for having multiple domains instead of a global index.

Figure 7 plots the overall cost of the InfoMatrix structure as a function of the domain size, for a 32K-node system. For domain sizes larger than 5000, we use the time-out mechanism for index-maintenance. Note that the right extreme, with the domain size being 32768, corresponds to using a global index. On this extreme, nodes process an average of 0.5 messages per second. On the other hand, with a domain size of 256, the average number of messages for partial-lookup queries and updates is only 0.05 per second, thus being an order of magnitude more efficient than the use of the global DHT. Even with all queries being total-lookup queries, a domain size of 512 is seen to require only about 0.10 messages per node per second, which is only one-fifth the cost of maintaining and using a global index.

Query Latency: We defer the evaluation of the latency experienced by queries to Sections 4.5 and 5.1.

Scalability: We evaluate how the InfoMatrix scales as the number of nodes in the system increases dynamically in Section 5.1. We simply note here that the overall overhead of the InfoMatrix remains between a factor 5 and 10 lower than that of a global index, even as the number of nodes varies from 4096 to 1 million.

To illustrate scalability with respect to the number of tuples per node, we consider a 32K node system with nodes having to index an average of 3000 keywords each, which is a factor 10 higher than that in our prior experiments. The overall cost of a global index goes up from 0.51 messages/second to 4.7 messages/second (a near-linear cost increase); on the other hand, the overall cost for the InfoMatrix, with a domain size of 256, remains almost constant at 0.06 messages/second.

3.5 Summary

We have seen that the use of a global index leads to high index-maintenance costs, together with a relatively low query precision. On the other hand, the use of the InfoMatrix allows the creation of smaller domains, enabling a reduction in the overall cost of queries and index-maintenance. In addition, we observe that the InfoMatrix offers higher-precision queries, ensuring a more uniform distribution of index-maintenance costs across nodes, and scaling well with an increase in the number of tuples per node.

4. DEALING WITH THE PHYSICAL NETWORK: THE SECOND CUT

So far, we have assumed that the physical network is ho-

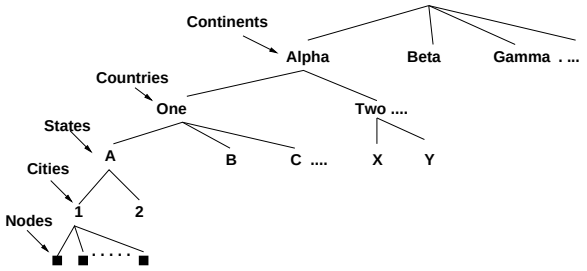

Figure 8: Nodes organized in a conceptual hierarchy

mogeneous and that it is equally expensive for every pair of nodes to communicate with each other. In reality, nodes are distributed over a heterogeneous physical network, and we therefore need to optimize for this network to reduce query latency and minimize bandwidth usage. Furthermore, we would also like query results to have *nearest-k* semantics, as discussed earlier. We begin by discussing a hierarchical model that captures the proximity of nodes on the physical network, and then explain how the inter- and intra-domain InfoMatrix design takes advantage of this model.

4.1 A Hierarchical Network Model

The participant nodes in the InfoMatrix can be organized in a "conceptual hierarchy" that captures the nearness of nodes on the physical network. For example, Figure 8 groups nodes by the city they are in, then groups them further by the state they belong to, then by country, and finally by continent. We then postulate that the *network distance* between two nodes is simply the distance between the nodes in this conceptual hierarchy. For example, any two nodes in the same city are at distance 2 from each other, while a node in country One is at distance 8 from a node in country Two.

Of course, this hierarchy is merely intended to serve as an example and is not reflective of the actual internet structure. However, it is well-known that the internet is hierarchically structured [25], and therefore, it is possible to reasonably capture the distance between pairs of nodes by a simple hierarchy like the one we have shown here.

Our goal is to ensure that as much communication as possible, both for queries and for index maintenance, occurs between nearby nodes on the physical network , i.e., nearby nodes on the hierarchy. As the first step, we define domains simply to be one level in this hierarchy of nodes. For example, if domains are defined at the "state" level, each state in the world corresponds to a unique domain. Consequently, there is a conceptual hierarchy, both on top of the domains (country, continent), and within the domain itself (city). We refer to levels in the hierarchy above the domain level as *superdomains*, and levels below as *subdomains*.

We now discuss how the inter-domain and intra-domain structures are modified to exploit this hierarchy. In the rest of this discussion, we assume that each node has a hierarchical name reflecting its position in the hierarchy. For example, a node in city 1 in state A in country One in continent Alpha is assigned the hierarchical name *Alpha.One.A.1*. We will assume that each node knows its hierarchical name. We do not discuss exactly how this may be achieved, except to point out that there are several known techniques for identifying the location of nodes on a physical network.

For example, one solution is to establish "beacons" spread around the network; each node measures its physical-network distance to these beacons, and chooses a hierarchical name based on the ordering of the beacons from the nearest to the furthest [13]. An alternative is to use services that map IP addresses to geographic location, and define domains on the basis of geography.

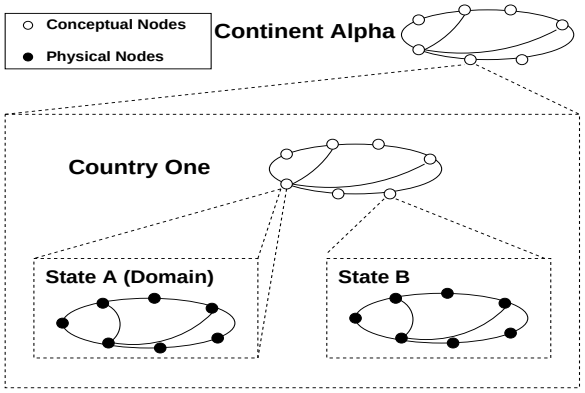

Figure 9: The interconnection of domains

4.2 The Inter-domain Structure

Let us return to our example, with domains corresponding to each state. In our original inter-domain design, we would simply have organized all the states in a conceptual Chord structure, and performed iterative broadcast on this structure. However, such a solution fails to exploit network-proximity information: states within a country are closer to each other than an arbitrary pair of states and, so, a query initiated in state A in country One should first be propagated to other states in the same country, before being sent to states outside the country. Similarly, the query should be propagated to other countries within the same continent, before being sent to other continents. We now describe both a modified interconnection network, and the iterative broadcast algorithm that exploits such proximity.

4.2.1 The Interconnection Network

Our inter-domain interconnection network consists of conceptual Chord structures at each level of the inter-domain hierarchy. Figure 9 depicts this organization for the lower two levels of our hierarchy. (The top-level interconnection of continents is not shown.) At the lowest level, the different states within each country are interconnected in a Chord structure, just as described in Section 3. The figure shows the different states in Country One, being interconnected, with each state represented by a hollow circle. The ID of each state in this structure is obtained by hashing the hierarchical name of that state.

At the next level of the hierarchy, the countries within each continent are again interconnected in a Chord structure, just as at the lower level. The ID of each country is again obtained by hashing the hierarchical name of that country. The top structure in the figure reflects this interconnection of countries. Of course, recall that links between countries are conceptual; a conceptual link from country A to country B is implemented by requiring each node in country A to link to any node in country B. At the top level of the hierarchy, the set of continents are arranged in yet another Chord structure. Again, the conceptual inter-continent links are implemented via multiple inter-node links.

Each node in the system establishes a number of inter-domain links, some serving as inter-state links at the lowest level of the inter-domain hierarchy, some inter-country links at the next level, and some inter-continent links at the highest level. The following theorem shows that the total number of inter-domain links going out of any node in the system remains logarithmic in the total number of domains.

THEOREM 5. *Given a fixed-height hierarchy consisting of k domains, the out-degree of any node in the system is $O(\log k)$ with high probability.*

4.2.2 Query Propagation

Query propagation on our hierarchical inter-domain structure is similar to query propagation in our flat inter-domain structure, and can be seen as being repeated invocations of Algorithm 1 (*FindQueryAnswers*), using the Chord networks at progressively higher levels of the hierarchy.

Say node s in state A desires to propagate query q. Initially, query propagation occurs by a direct execution of Algorithm *FindQueryAnswers* within country One. Thus, query q would be executed within state A, and then be propagated to other states in country One, using the Chord structure interconnecting these states.

If not enough answers are found, node s "steps up" a level, and propagates q to other countries using the Chord interconnection of countries. This propagation is again achieved using Algorithm *FindQueryAnswers*, except that the Chord network is on countries instead of states. The sole difference is in Algorithm *PropagateQuery*, where each node receiving the query now needs to propagate it not only to other countries, but also to all states within its own country.

For example, consider when node s sends the query to some node t in country Two, and asks t to propagate the query further to other countries. Node t can propagate the query to the desired set of countries just as earlier, using the inter-country Chord network. However, in addition, node t is also responsible for propagating the query to all states within its own country. Performing this propagation is simple: node t simply broadcasts the query on its own inter-state Chord structure using Algorithm *PropagateQuery*, with all answers being returned to node s.

Similarly, if s does not find enough answers, even after exhausting all countries in continent Alpha, it switches to the inter-continent structure. Again, a node receiving the query at the inter-continent level is responsible not only for propagating the query to other continents, but also for propagating it to all states within its own continent. We omit a more detailed algorithm description due to space constraints.

4.3 The Intra-Domain Structure

Recall that queries within a domain are answered by *routing* the query to the node within the domain that maintains the hash bucket relevant to the query. Ideally, this query routing should take place in a manner that respects the intra-domain hierarchy. For example, with each state being a domain by itself, a query initiated at a node n_1 in city 1, and destined for another node n_2 in the same city, should never have to pass through a node in any other city, even in the same state. This locality property is important both for efficiency, and for good fault isolation.

Our solution in Section 3.1–using plain Chord as the intra-domain structure–does not offer such locality properties. In a parallel work soon to be published, we describe a hierarchical construction based on Chord, called Crescendo, that provides these locality properties, as well as a variety of other properties that will prove important in Section 5.

Note that the exact manner of link creation in Crescendo is *different* from the link creation mechanism for our inter-domain structure, despite the fact that both these structures exploit the hierarchical classification of nodes. However, the details of the link creation do not concern us, and we summarize the following key points about Crescendo.

- Crescendo and Chord provide similar routing properties, using $O(\log n)$ links per node to achieve routing in $O(\log n)$ hops.
- Crescendo provides path locality. If a node s routes to a destination t, the message never leaves the portion of the hierarchy containing both s and t. Thus, routing is efficient with respect to the physical network.
- Crescendo possesses recursive structure. The Crescendo structure interconnecting nodes within a state contains Crescendo structures that interconnect nodes within each city.

4.4 The Overall Structure

We have now defined the overall InfoMatrix structure: Nodes continue to be divided into domains, and there is a conceptual hierarchy both within each domain, and on top of domains. Both the intra-domain and inter-domain structures optimize for the corresponding conceptual hierarchy. The bootstrap mechanism for new nodes joining the system operates just as described in Section 3.3, but on the hierarchical inter-domain structure, instead of the flat structure. Each node needs to know (a) its hierarchical name, and (b) some existing node in the system, for it to join the system.

All the theorems stated in Section 3 continue to hold in this modified hierarchical structure. We show in our evaluation that this InfoMatrix structure that exploits multi-level hierarchies, helps improve query latency, and reduces the system bandwidth consumption.

4.5 Evaluation

We now evaluate the efficacy of the InfoMatrix's adaptation to the underlying physical network. In order to do so, we use the standard GT-ITM [25] model of internet structure to produce a so-called transit-stub topology: a hierarchical interconnection of routers modelling the structure of the internet. Nodes are attached uniformly at random to each "stub" router in this GT-ITM topology, and are classified in a four-level hierarchy on the basis of this physical network structure. Domains are defined to be at the third level of the hierarchy, resulting in the formation of 400 domains. We perform experiments, just as earlier, on a 32K-node system based on this hierarchical structure.

Query Latency: Figure 10 shows the cumulative distribution of the latency to the 20^{th} result of a partial-lookup query[4], both for our network-aware hierarchical InfoMatrix structure, and a network-oblivious, flat InfoMatrix structure with the same number of domains. We also use two vertical lines to show the average latency of lookups in a global-index structure, both optimized for the underlying network using Crescendo, and using plain Chord to build the index.

[4]For queries with less than 20 total results, we use the latency of performing the iterative broadcast to the entire system.

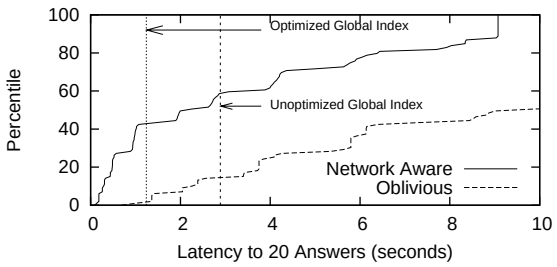

Figure 10: Distribution of partial-lookup latencies in optimized and unoptimized InfoMatrix

Cost (#msgs/ node/sec)	InfoMatrix		Global Index	
	Optimized	Unopt.	Optimized	Unopt.
Query	0.67	0.84	0.05	0.09
Update	0.18	0.65	4.19	4.19
Total	0.85	1.49	4.24	4.28

Table 1: Number of network links used per node per second, for the InfoMatrix and a Global Index

First, we observe that the use of the multi-level hierarchical structure offers a great latency improvement in the InfoMatrix. The median query latency (and the average) on the optimized InfoMatrix is about 2 seconds, while the maximum latency is about 9 seconds. On the other hand, the unoptimized InfoMatrix has a median (and average) latency of 10 seconds, while the maximum latency is as high as 25 seconds (not shown in figure). Second, nearly 43% of queries have a lower latency on the optimized InfoMatrix, compared to an optimized global index. (Although our figure does not show the exact distribution of latencies for the global index, this is indeed true.) For the remaining queries, the optimized global index offers lower latency, but we believe it is reasonable to trade off on the latency of queries with very few results, in order to reduce the overall system overhead (as seen in Section 3.4, and will be seen next).

Network Bandwidth: Next, we attempt to quantify the benefit of our network adaptation in terms of savings in physical-network bandwidth. We use a simple metric– the number of physical-network messages sent per node per second, both for queries and index-maintenance–to evaluate the optimized and unoptimized versions of the InfoMatrix as well as a global index. (Note that a message from one node to another that traverses 5 physical-network links will be counted as 5 messages.) Table 1 presents these results. We see that the optimized version of the InfoMatrix uses more network messages than the global index for queries, but more than makes up for it in index-maintenance cost, ensuring that the overall number of messages used by it per node per second is a factor 5 smaller than the number used by the global index. We also see that the unoptimized version of the InfoMatrix uses more than 1.7 times as many messages as the optimized version.

Nearest-k semantics: Finally, we observe that the use of the network hierarchy results in *nearest-k* semantics for partial-lookup queries, meaning that queries tend to return results that are physically close to the querying node. We do not quantify this property in this version of the paper.

5. DYNAMIC DOMAIN CHOICES

So far, we have assumed that the set of domains is static. However, it may be important to allow the number of domains to itself change over time. For example, if there are 500 nodes per state, we might treat all nodes within a state as being a single domain. However, if the number of nodes in a state increases over time to, say, ten thousand, it may be too expensive to maintain an intra-domain index over the content of that many nodes. Therefore, we might want to *split* the state into multiple domains, and build indexes within each of the new domains. Similarly, if the number of nodes in a state is particularly small, we might want to group together multiple states into a single domain so as to create a larger index and avoid having to broadcast a query to each of the states.

A simple way to model both this splitting or merging of domains is to interpret them simply as *lowering or raising* the level in the hierarchy at which the domain is defined. Of course, there is no requirement that all domains be at the same level in the hierarchy. It is perfectly possible for some domains to be at the state level, some other domains to be entire countries, while other domains are confined to individual cities within a state. For example, when a domain at the state level, say state A, becomes too large, this domain is eliminated and each city in state A becomes a domain instead[5]. Other domains outside state A remain unaffected by this splitting of state A.

Note that the splitting and merging of domains necessitates modifications to the intra- and inter-domain interconnection network, as well as to the organization of indexes in the domains involved. It turns out that our careful hierarchical construction of intra-domain and inter-domain structure ensures that there needs to be very little modification to the interconnection network, and only minor re-organization of data, when domains split and merge as described above. Due to space constraints, the exact description of this process is left to the extended version of this paper, and we only state the following theorem characterizing the costs of domain splits and merges.

THEOREM 6. *When a domain D splits into s domains (or when s domains merge to form a single domain D), each node in D needs to send only $O(s)$ messages to reorganize the InfoMatrix, irrespective of the number of nodes in D. Nodes outside D are not involved in the operation.*

There are two questions left: (1) When should domains split and merge? (2) How do nodes in a domain coordinate to split and merge at the same time? There are many different options available for question (1). We choose one of the simplest: a domain splits when its size exceeds a fixed constant, say 1024. Domains merge when their cumulative size is less than a different constant, say 512.

For question (2), it turns out that there is no coordination necessary, and each node can individually decide what the domain "level" ought to be at any instant of time. Our hierarchical inter-connection structure ensures that queries and updates are executed correctly even when nodes have conflicting "beliefs" about what the domain level is. We again omit details.

[5]In order to enable arbitrary numbers of domain splits, the hierarchy is defined to extend downwards to arbitrary depth, by appending an "infinite" binary hierarchy below the lowest level of the network-based hierarchy (the city level in our example). Nodes are assigned at random to different branches of the binary hierarchy.

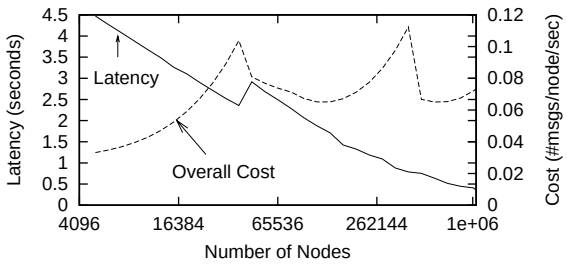

Figure 11: Overall Cost and Query Latency as the number of nodes increases

5.1 Evaluation

We now evaluate how the InfoMatrix, with dynamic definition of domains, scales as the number of nodes in the system increases continuously. Figure 11 plots the latency to the 20^{th} result for partial-lookup queries, as well as the overall cost of executing queries and maintaining the index, as the number of nodes in the system increases continuously from 4096 all the way up to 1 million. Nodes use the same multi-level network hierarchy defined in Section 4.5, and domains are constrained to never be larger than 1024 nodes.

We first note that the latency of partial-lookup queries declines on a nearly continuous basis, from about 4.5 seconds in a 4096-node system, down to under 0.5 seconds in a million-node system, thus scaling extremely well with the number of nodes in the system. This is not surprising since, for most queries, there are far more answers available in a million-node system than in a smaller system. There is a slight increase in lookup latency when the number of nodes hits about 40000; this increase is due to the splitting of many domains, resulting in the formation of more domains and a consequent increase in query cost.

The overall cost, in terms of the number of messages per node per second, is also seen to scale well with the number of nodes, never exceeding 0.12 messages per second. We observe two "peaks", both corresponding to points where the number of domains increases due to domain splits. As we move from one peak to the next, the valley reflects the sum of a steadily declining query cost and a steadily increasing index-maintenance cost. When the next peak is reached, the domains split, dropping the index-maintenance cost down again, while increasing query cost.

We have not shown the cost of total lookups, as it increases almost linearly with the total number of nodes in the system. This is not surprising, since we chose to limit the size of a domain to control the update cost. In contrast, a global index scales more gracefully for total-lookups, increasing its cost only logarithmically with the number of nodes. However, we believe this increased cost of total lookup is acceptable; our partial-lookup curves do account for the cost of the small fraction of highly selective queries, and still retain a low overall system cost that is a factor 5 to 10 lower than the overall cost of a global index.

6. EXTENSIONS AND CONCLUSIONS

So far, we have only discussed the utility of the InfoMatrix for simple selection queries with an equality predicate, in scenarios where the cost of index-maintenance is a problem. However, this is only one piece of the puzzle that the InfoMatrix architecture is designed to solve. We now describe both the additional capabilities offered by the InfoMatrix, and application scenarios where these capabilities make the InfoMatrix preferable to a global index structure.

6.1 Complex Relational Queries

The InfoMatrix is capable of supporting more complex relational queries. *Range queries* on a single attribute are supported by replacing the intra-domain hash-index structure by a new range-partitioned index structure that we describe in a concurrent, parallel work. *Multi-predicate* and *natural join* queries are supported in the InfoMatrix by the use of index intersection.

The support for multi-predicate queries exposes one of the advantages of the InfoMatrix organization, compared to a global index. For example, consider a multiple-keyword IR query in our file-sharing scenario, which can be answered by the use of index intersection. To explain, the list of tuple identifiers matching each keyword is retrieved, and intersected to find tuples matching all the keywords. Performing such index intersection using a global index is expensive, since long inverted lists need to be transmitted across the network [22]. Although it is possible to reduce the cost of this intersection by the use of bloom filters and semi-join-like techniques, the transmission of these inverted lists still remains an expensive operation.

The InfoMatrix's use of small-sized domains leads to much smaller inverted lists per keyword within each domain. When inverted lists are transmitted across nodes, communication is network-efficient since lists are small and nodes within a domain are physically close. For partial-lookup queries that only need to execute the query in a small number of domains, such a partitioning of the inverted lists results in savings in terms of index-intersection cost, while also providing *nearest-k* semantics. Moreover, with small domain sizes, each node is responsible for a larger set of keywords, leading to a larger fraction of queries for which no inter-node communication is necessary for intersection [22].

6.2 Multi-level Indexing

The InfoMatrix's use of a multi-level hierarchy allows the *simultaneous* use of all levels in the hierarchy as the indexing domain. For example, a node with highly stable content could index its content globally, treating the entire system as a single domain. A node with unstable content could index its content treating only the nodes in its own city as the indexing domain. Despite each node having this freedom to choose where its content is indexed, it is possible to efficiently and adaptively route and propagate queries to obtain the desired number of answers. Such a structure enables fine-grained trade-offs between query and index-maintenance costs on a per-tuple, or per-node, basis. More details on multi-level indexing are described in the extended version of this paper.

6.3 Constrained Indexing

In many applications, a node may not be willing to allow its content to be indexed by arbitrary nodes in the system, for reasons of privacy, access control, trust, copyright or just answer quality. We now present applications which may require such constraints for some of the reasons listed above, and explain how the InfoMatrix is useful for each of them.

Copyright: A distributed digital library may operate over a set of machines set up in public libraries across a

country. Public libraries within a county district may have their content indexed by each other's machines, but copyright limitations may prohibit the indexing of their content by other nodes in the P2P system.

Access Control: A photograph-sharing application may rely on a hierarchical access-control mechanism, which dictates what subtree of the hierarchy is permitted to view a certain photograph, to limit the set of nodes that may maintain an index entry for each photograph, and ensure that queries issued by a node return references only to tuples the node is permitted to access.

In both the above cases, the InfoMatrix's support of multiple, hierarchically structured domains (combined with simple access-control enforcement) can be used to support distributed indexing obeying such constraints, while still allowing seamless partial and total-lookup queries.

Privacy: Privacy-preserving indexing [10] requires that query results include carefully-chosen false positives, to ensure that it is impossible to prove that a specific node owns a specific tuple by the execution of any number of queries. Reference [10] suggests a privacy-preserving indexing scheme to build a central index, which can be naturally extended to create a distributed, privacy-preserving index using the multi-level InfoMatrix structure. The InfoMatrix also allows nodes to utilize the *query results*, false positives and all, to efficiently find tuples matching the query that the node is *authorized* to view.

6.4 Conclusions

The InfoMatrix improves on global indexes by using dynamic trade-offs between query cost and index-maintenance cost to reduce the overall cost in a P2P system. It scales well both with the number of nodes, and the amount of data per node, unlike global indexes, which suffer on the latter count. It adapts to the structure of the underlying physical network, optimizing query latency and bandwidth consumption, and offers nearest-k semantics for queries. Experiments based on real data suggest that it has an order of magnitude less overhead than a global index, while providing comparable query latency for partial lookups. The InfoMatrix offers other advantages in scenarios that require constrained indexing or complex queries.

7. REFERENCES

[1] Gnutella. Website http://gnutella.wego.com.

[2] Kazaa. http://www.kazaa.com.

[3] Overnet. http://www.overnet.com.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, pages 202–215, 2001.

[5] P. Ganesan, Q. Sun, and H. Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *Proc. IEEE Infocom*, 2003.

[6] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. ICDE*, 2003.

[7] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[8] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proc. VLDB*, 2003.

[9] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. PODC*, 2002.

[10] M.Bawa, J. R.J.Bayardo, and R.Agrawal. Privacy preserving indexing of documents on the network. In *Proc. VLDB*, 2003.

[11] M.Castro, M.B.Jones, A.M.Kermarrec, A.Rowstron, M.Theimer, H.Wang, and A.Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proc. IEEE Infocom*, 2003.

[12] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.

[13] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. IEEE Infocom*, 2002.

[14] S. Ratnasamy, S. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content addressable networks. In *Proc. 3rd Intl. Networked Group Communication Workshop (NGC 2001)*, 2001.

[15] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, pages 329–350, 2001.

[16] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, pages 188–201, 2001.

[17] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. 3rd Intl. Networked Group Communication Workshop*, 2001.

[18] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN'02)*, 2002.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, 2001.

[20] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. Technical Report TR-CIS-2003-01, Polytechnic Univ., 2003.

[21] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. ACM SIGCOMM*, 2003.

[22] A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in distributed text document retrieval systems. *The VLDB Journal*, 2(3), 1993.

[23] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proc. VLDB*, 2001.

[24] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer systems. In *Proc. ICDCS*, 2002.

[25] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, 1996.