Adlib: A Self-Tuning Index for Dynamic Peer-to-Peer Systems

Prasanna Ganesan Qixiang Sun Hector Garcia-Molina

Stanford University
{*prasannag, qsun, hector*}*@cs.stanford.edu*

Abstract

Peer-to-peer (P2P) systems enable queries over a large database horizontally partitioned across a dynamic set of nodes. We devise a self-tuning index for such systems that can trade off index maintenance cost against query efficiency, in order to optimize the overall system cost. The index, Adlib, dynamically adapts itself to operate at the optimal trade-off point, even as the optimal configuration changes with nodes joining and leaving the system. We use experiments on realistic workloads to demonstrate that Adlib can reduce the overall system cost by a factor of four.

1 Introduction

A peer-to-peer (P2P) system consists of a large, dynamic set of computers (*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. We focus on systems where nodes contain "related" data, i.e., we can view each node as "owning" some tuples in a global, horizontally partitioned relation R.

A fundamental operation in such a P2P system is a selection query that requires all, or some, tuples that have a "key" attribute A equal to a given value. Such a query may be answered efficiently using a distributed index that maps each possible value of attribute A to the set of nodes that contain tuples with that value. There are many different ways of constructing such a distributed index, each of which offers a different trade-off between the cost of constructing and maintaining the index, and the cost of using the index to answer queries.

To illustrate, consider two common indexing structures that have been proposed in past literature. In the Gnutella-style [1] approach, each node constructs a local "index" over the tuples that it owns itself. Therefore, a query for all tuples with a given attribute value needs to be sent to all nodes in the system, resulting in a high query cost. On the other hand, index maintenance is free. When a new node joins the system, or an existing node leaves, the indexes of other nodes are completely unaffected.

A second approach is to construct a distributed global index, partitioned across nodes by attribute values; for each value, some one node in the system is designated to manage and store the entire index entry – a list of all nodes with tuples containing that value. The assignment of which nodes manage which values may be done in different ways, for example, by hash partitioning [18] or by range partitioning [6]. Such a global index offers efficient querying; a query for a given value is answered simply by contacting the node managing the index entry for that value. On the flip side, every time a node N joins or leaves the system, all nodes that manage index entries for values owned by N need to be notified, in order to update the index appropriately.

Many P2P systems are characterized by query rates that are comparable to the rates at which nodes join and leave [17, 19]. In such systems, the index-maintenance cost of the global-index approach can be very high, and dwarf the benefit obtained for queries. Worse still, as the systems scale up in terms of the amount of data owned by each node, the index maintenance cost grows much faster than the cost of queries, rendering the global index expensive.

Our objective is to devise a self-tuning index 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 the scheme to scale well with the amount of data owned by each node, since we expect the data volume per node to grow rapidly over time, even more so than the number of participant nodes in the system.

We now illustrate the intuition behind our solution, Adlib, which offers the above desiderata. At its heart, Adlib can be viewed as a two-tier structure. Nodes are partitioned into independent *domains*; nodes within a domain build a distributed "global" index *over the content stored in that domain*. Thus, all query answers within a domain may be

obtained by contacting just one node in that domain. Finding all answers to a query entails contacting one node in each domain, which is fairly efficient if the number of domains is small. When a node joins or leaves the system, only other nodes within its domain need to be notified, thus limiting the index maintenance cost. In Section 2, we provide a high-level overview of data storage, index construction and query routing in this basic architecture.

The first issue arising from the above architecture is to identify the right number of domains to use. In Section 3, we develop a cost model to analyze the costs of queries and index maintenance, and characterize the optimal number of domains needed to minimize the total cost of queries and index maintenance. We validate our model by means of experiments with realistic data, and show that our architecture offers the opportunity to reduce the overall system cost by a factor of four, compared to earlier solutions.

From our analysis and experimental results, we show that the optimal number of domains is a function of the number of nodes in the system. Consequently, the number of domains has to change dynamically with the network size. In Section 4, we introduce practical indexing algorithms and overlay networks that enable dynamic, self-tuning splitting and merging of domains . A key challenge we address is to ensure that the algorithms are fully distributed, require no centralized locking and synchronization, and allow index reorganization while simultaneously supporting queries in the system. We show experimentally that our algorithms operate well and provide good performance even as the size of the network changes over time.

The Adlib index also addresses challenges arising from the heterogeneity of the underlying physical network. Since the latency of communication between different nodes in the system may be different, we need to optimize query and index-maintenance traffic for such a physical network. We discuss such network optimizations in Section 5.

2 The Adlib Architecture

We now describe the high-level architecture of the Adlib index. For now, assume that there is a fixed number of domains k. Each node is assigned to one of these k domains when it joins the system, for example, at random. Queries may be issued by any node in the system and are of two types. A *total-lookup* query requires all tuples that have a specified value for the search key. A *partial-lookup* query requires any P tuples that have the specified value for the search key, for some constant P. If the number of tuples with that value is less than P, the partial-lookup query is equivalent to a total lookup.

Adlib uses a two-tier structure – intra-domain and inter-domain – to index content and execute both total- and partial-lookup queries. We now discuss the basic design of each of these structures in turn. In Section 4, we will see how these structures are constructed and made self-tuning.

2.1 The Intra-domain Structure

Nodes within each domain construct a distributed index over the content in that domain, partitioning the index using hash or range partitioning [6]. For concreteness, we describe a hash-partitioning scheme called consistent hashing [12]. Each node in a domain chooses a unique intra-domain ID, at random, from a large, circular space of IDs. Values for the search-key attribute A are also hashed to the same space using a hash function; a node stores index entries for all values that hash to a number between the node's ID and the next larger node ID in the domain. Recall that the index entry for a value is a list of all nodes in the domain that have a tuple with that value.

Nodes within a domain are interconnected in an *overlay network* that enables any node to *route* a query q to the manager of the relevant index entry for q. We may use an overlay network such as Chord [18], that allows a query to be routed to its destination using only $O(\log d)$ inter-node messages, where d is the number of nodes in the domain. Thus, a node may initiate a query q and find all answers within the domain using this structure.

When a new node joins the domain or an existing node leaves, three operations need to take place: (a) the overlay network structure needs to be modified suitably, (b) index entries need to be re-distributed across nodes to allow the new node to hold some entries, or to make up for the entries lost by the leaving node, and (c) the index needs to be updated so that the new node's *content* is indexed, or that the old node's *content* is removed from the index.

Operation (a) is well-understood, and it is well-known [18, 16] that the overlay network can be updated efficiently as nodes join and leave, using just $O(\log d)$ messages per join or leave. (In contrast, if each node knows every other node in the domain using a replicated directory, updates for node joins and leaves would have to be sent to every node in the domain, which is very expensive.)

Operations (b) and (c) have received less attention in past literature, and we focus on them. When a new node m joins the domain, (i) it takes over some index entries from an adjacent node, and (ii) 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 overlay structure. Both these operations are straightforward. Updating the index when a node leaves is more interesting. We now describe two approaches to this problem.

The Time-Out Mechanism The traditional approach to index updates is the time-out mechanism [11]. Each inserted index entry has a time period, say T_o seconds, for which it stays alive. After this period, it "times out" and is deleted 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_o seconds. (We assume that refresh messages can be sent directly to the destination node instead of having to be routed along the overlay network.) When a node m leaves, index entries for its content will not be refreshed, and will therefore be deleted within T_o seconds. Nodes that attempt to contact m to refresh entries managed by m will realize that m has left, and may reinsert the index entries into the system.

The size of the time period T_o offers a trade-off between the cost of maintaining the index and *query accuracy*. If T_o is small, the maintenance cost is high since nodes have to refresh index entries frequently, but there are very few query results that refer to data no longer in the system (few false positives) and nearly all available results are returned by the query (few false negatives). If T_o is large, the maintenance cost is low, but query results may be less accurate due to false positives stemming from stale index entries to non-existent data, and false negatives due to the loss of index entries for data available in the system.

Observe that the maintenance cost for a node increases linearly with the number of tuples stored by that node, when the number of tuples is small compared to the domain size. This means that the maintenance cost does not scale well as the amount of data per node increases.

The Update-Broadcast Mechanism The relatively small number of nodes in an Adlib domain offers an alternative approach for index maintenance that has not been studied in the P2P literature. Whenever a node N leaves the domain, its successor – the node with the next larger ID – broadcasts the information about N's departure to all the nodes in the domain. This broadcast may be achieved efficiently on the overlay network, and requires only $d-1$ messages in a d-node domain. We present details in Appendix A. Each node receiving the broadcast then eliminates index entries for N's tuples. (If some of its own content is indexed at N, the node also re-inserts index entries for that content subsequently.) 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 "heartbeat" messages between adjacent nodes. Overlay networks already require such message exchange between successive nodes for their maintenance [18, 5, 11]; therefore, the heartbeats do not create an additional overhead. The periodicity of heartbeats governs the delay in updating the index to eliminate stale entries and, consequently, the query precision. Typically, heartbeats are very frequent (we assume at least one a minute), and we will see that the query precision is consequently very close to 100%.

Update broadcast offers two potential advantages over the time-out mechanism. First, the cost of broadcasting the failure of a node is *completely independent of the number of tuples owned by each node* – being only a linear function of domain size – and allows the system to scale up well 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 relatively uniform. Second, there is little additional overhead for index update, when index entries are replicated across multiple nodes to deal with node failures [18, 5]. We will compare and contrast time-outs with update broadcast in Section 3.2.

2.2 The Inter-domain Structure

So far, we have seen how a node may find all query answers within its own domain using the intra-domain index and overlay network. Executing a partial- or total-lookup query requires nodes to be able to gather results from some or all domains in the system respectively. We now describe the intuition behind the construction of an inter-domain overlay network, and a query propagation algorithm, that enables such queries.

Our solution relies on iterative broadcast, and has a simple intuition. A node first finds all answers to a partial-lookup query within its own domain. If the number of answers proves insufficient, it attempts to find answers from 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 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 sent to all the domains.

In order for the above approach to work, nodes need to maintain links to other nodes outside their own domain. We defer a detailed discussion of this inter-domain interconnection structure to Section 4. Here, we simply note that, if the total number of domains is k, each node maintains links to $O(\log k)$ nodes in *other* domains in order to support query routing. A query may be sent to f domains using less than $2f$ inter-domain messages.

We summarize the intra- and inter-domain overlay structure with the following theorem describing the number of links established by each node, and the cost of queries. Proofs are in Appendix B.

Theorem 1. *If there are n nodes in the system distributed uniformly across k domains, the following statements hold with high probability:*

(a) the total out-degree of each node is $O(\log n)$,

(b) the number of messages exchanged to handle a node join or leave is $O(\log n)$,

(c) a total-lookup query initiated by any node takes $O(k + \log(n/k))$ messages to obtain all answers,

(d) a partial-lookup query that contacts f domains takes $O(f + \log(n/k)$ messages to obtain answers,

(e) if the latency of a message transmission is one time unit, a total-lookup query is answered in $O(\log n)$ time units, and a partial-lookup query in $O(\log^2 k + \log(n/k))$ time units.

3 Optimizing Costs in Adlib and Validation

Having seen the basic Adlib architecture, we now devise a simple model of the costs of indexing and queries in this architecture. From this model, we identify the optimal number of domains to use in order to minimize the total system cost. We then validate our model and analysis with an experimental evaluation of the architecture on realistic data.

3.1 Modeling and Optimizing Costs in Adlib

Say nodes have t tuples each on average, have an average lifetime of L seconds, and an average query rate of Q queries per second. We assume nodes are uniformly distributed across k domains, i.e., each domain has n/k nodes. Throughout our analysis, we will ignore integer round-off errors: we assume n/k is an integer, that L is a perfect multiple of the time-out period, and so on.

Update Cost Recall that there were two alternatives for updating indexes: the time-out mechanism and update broadcast. We now characterize the costs of both these mechanisms.

Theorem 2. *(a) The cost of index creation and maintenance with update broadcast is $(3t/L)(1 - k/n) + n/(Lk)$ messages per node per second.*

(b) For a time-out period T_o, the cost of index creation and maintenance with time-outs is $(3t/L)(1-k/n) + \frac{n}{kT_o}(1 - (1 - k/n)^t)$ messages per node per second.

Corollary 1. *For a time-out period T_o, the time-out mechanism is more efficient than update broadcast if and only if $(1 - k/n)^t > 1 - T_o/L$.*

We see that the cost of the time-out mechanism is dependent on the time-out period T_o. If T_o is very large, the time-out mechanism is more efficient than update broadcast. However, a large T_o also implies that query accuracy is extremely low, whereas update broadcast always offers high accuracy. For reasonable values of T_o (as we will derive from our experiments), we can see from the formula that the time-out mechanism will be more efficient only if t is very small, or n/k, the size of a domain, is very large. Experimentally, we will see that update broadcast is more efficient upto a domain size of 5000 even when each node has as few as 300 tuples. Consequently, we focus the rest of our analysis on update broadcast.

Query Cost Modeling the cost of total-lookup queries is straightforward. However, coming up with an analytical formula to estimate the cost of partial-lookup queries requires a model of how content is distributed in the network. For the purposes of analysis, we postulate the following model of content distribution: The set of tuples owned by each node is drawn from a fixed universe of tuples, and is independent of the total number of nodes in the system.

The above postulate is reasonable, since it merely says that the content of a node is independent of the system size. A partial-lookup query requires a fixed number of answers and needs to examine some number of nodes, say N_p, before it is satisfied. We model the execution of the partial lookup as examining nodes one by one in random order until the query is satisfied.

By our assumptions, it follows that the random variable N_p is dependent only on the value being queried for and the distribution of content, and is completely independent of the number of nodes in the system. Therefore, the number of nodes to be examined by a partial-lookup query, averaged over all queries, is a constant. We denote this constant number of nodes as C.

The costs of partial- and total-lookup queries can then be modeled as follows:

A total-lookup query requires $1.5k + 0.5\log(n/k)$ messages to find all answers, while the average partial-lookup query requires $3Ck/n + 0.5\log(n/k)$ messages to find sufficient answers.

Optimizing Costs Having modeled both update and query costs, we can now identify the optimal domain size in order to minimize the cumulative cost of queries and updates. If all queries are total lookups, we see that the total cost per node per second is $(3t/L)(1 - k/n) + n/(Lk) + Q(1.5k + 0.5\log(n/k))$, which is minimized when $k \simeq \sqrt{2n/(3QL)}$. Thus, both the number of domains and the domain size should be $\Theta(\sqrt{n})$, when queries are total lookups. (If the query rate is extremely high, $\frac{n}{QL}$ may become less than 1, in which case the optimal solution is to just use a single domain.)

If all queries are partial lookups, the total cost is minimized when $n/k \simeq \sqrt{3CQL - 3t}$. In other words, the optimal domain size is independent of n, and dependent primarily on the number of nodes desired to be reached by a partial lookup, the query rates and lifetimes ($CQL >> t$ typically, so the dependence on t is very weak); consequently, the optimal number of domains is directly proportional to n. If queries are a mix of partial and total lookups, we may once again derive the optimal number of domains to be roughly linear in $\sqrt{n}$, but slightly less than in the total-lookup case.

In summary, the optimal number of domains is proportional to $\sqrt{n}$ for total-lookup queries, and proportional to n for partial-lookup queries.

3.2 Validation

We now evaluate the Adlib architecture, and our analysis of the optimal domain size, using real data gathered by Saroiu et al. [17] 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. We treat each keyword in a file name as a tuple owned by the node, after eliminating stop words like "the" which are ubiquitous.

On cleaning the data in the above fashion, each node owns $t = 307$ tuples on average. Node lifetimes follow a skewed distribution, with a mean lifetime of 3 hours, and a median lifetime of 54 minutes. We extrapolate this data to simulate larger systems with n nodes, for arbitrary n, assuming that each node's lifetime characteristics, and set of tuples, follows the same distribution as that of the measured data. Note that our extrapolation is limited by the fact that we cannot create "new" tuples when increasing the number of nodes beyond 4000; we only end up replicating content already present in the system, which helps increase the success rate of partial-lookups. However, this limitation does not affect our evaluation of total-lookup queries.

Our query workload is obtained from a study of the Gnutella network by Yang et al. [20] which gathered a trace of 100,000 queries being executed on the network; we treat each keyword as a separate query issued in the system. We obtain the query rate from a different study by Yang et al. [19] on the OpenNap system, which suggests that the rate is 0.00083 queries per node per second (one query per node per 20 minutes).

Simulation Setup We simulate an n-node Adlib with k domains, assigning each node to a random domain. Nodes join and leave the network, with the lifetime of nodes being drawn from the measured distribution of lifetimes. Each node leave is accompanied by a new node join, to ensure that the total number of nodes is always around n. Nodes issue queries at a uniform rate (of one query every 20 minutes). When a node joins or leaves, we allow the overlay network to adapt "instantly" to set up the appropriate links. We thus do not simulate the effects of inconsistencies in the overlay network on query performance; we believe this is acceptable, and even desirable, since our focus is on evaluating *data-centric* effects and costs, rather than on the robustness of the routing network structure. We assume no messages are lost in communication; again, we believe such an assumption is appropriate in this context.

A. Update Costs

Our first undertaking is to evaluate the relative costs of index maintenance with time-outs and update broadcast. We first quantify the trade-off offered by the time-out mechanism between query accuracy and the cost of periodic refresh messages, for different values of the time-out period T_o. This trade-off is independent of the number of nodes in the system. Figure 1 depicts the fraction of stale answers, i.e., the fraction of false positives in the query results, as a function of T_o. We see that if T_o is one hour, more than 40% of the answers are stale. In order to achieve a staleness of under 10%, T_o needs to be smaller than 10 minutes. (Since lifetime measurements are made at coarse granularity in our data set, it is hard to accurately determine the time-out period necessary to achieve lower values of staleness.) We do not show the effect of T_o on false negatives, as it is similar.

Figure 2 shows the cost of update broadcast, as well as time-outs (with $T_o = 10$ minutes), for different domain sizes. (We do not depict the costs of inserting and re-inserting index entries, since they are the same for both schemes.) 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 the formula in Section 2.1, plugging in the appropriate values for the relevant parameters. 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*.

Note that the fraction of false positives using update broadcast (not shown in figure) is less than 1%, compared to the 10% of the time-out mechanism. Also note that the cost depicted for the time-out mechanism is the *average* cost per node; the maximum cost for a node is a factor 16 higher. Finally, if the system content scales up to $t = 3000$ tuples per node, the update cost with timeouts increases nearly ten-fold, while the cost of update broadcast remains almost the same. We conclude, therefore, that update broadcast is better than time-outs for reasonably sized domains, and is likely to become more desirable as the data per node increases.

B. Query Cost

Figure 3 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 lookups, and partial lookups which are terminated after finding the first 20 query answers. 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). 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 enough answers available for many queries, so that it is sufficient to use small domain sizes and query only a

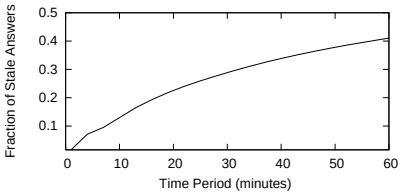

Figure 1. Fraction of stale results as a function of time-out period

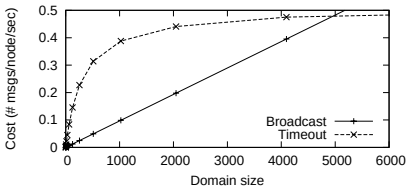

Figure 2. Update cost as a function of domain size

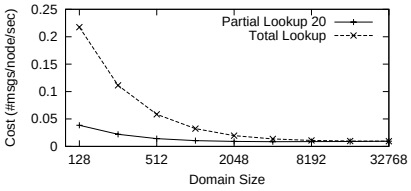

Figure 3. Cost of total and partial lookup queries vs. domain size

small number of domains.

C. Overall Cost

Having seen that update costs increase with domain size while query costs decrease, we now show the *overall cost* of queries and index maintenance in order to understand the total system overhead in maintaining the index and executing queries. We define the overall cost as the sum of the total query cost, the index-maintenance cost (including insertion, re-insertion and movement of index entries), 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.

Figure 4 plots the overall cost of the Adlib structure as a function of domain size, for a 32K-node system. For domain sizes larger than 5000, we use the time-out mechanism for index maintenance to present the best possible overall cost for each domain size. 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 0.6 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 0.15 per second, thus being four times as efficient as the use of a global index. Even with all queries being total-lookup queries, a domain size of 512 is seen to require only about 0.2 messages per node per second, which is only one-third the cost of a global index.

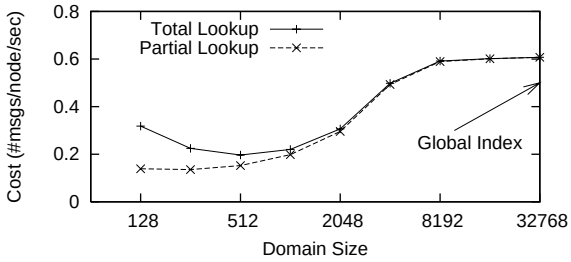

Figure 4. Overall cost as a function of domain size

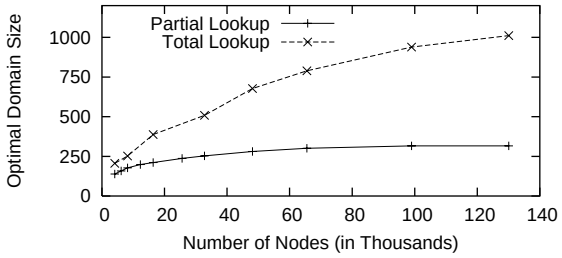

Figure 5. Optimal Domain size as a function of n

D. Varying the Number of Nodes

Next, we attempt to identify the *optimal* domain size as a function of the number of nodes n. For each value of n we consider, we run multiple experiments, each with a different value of domain size, in order to identify the optimal domain size for both total and partial lookups. We then plot this domain size as a function of n in Figure 5.

For total lookups, we observe that the optimal domain size is almost exactly proportional to $\sqrt{n}$, as predicted by our model. (Observe that the optimal size is nearly 500 for $n = 32768$ and increases to 1000 when n quadruples.) For partial lookups, the optimal domain size initially creeps up slowly with n, but flattens out as the number of nodes increases. This result is fairly consistent with our model in Section 3, which suggests that the ideal domain size is a fixed constant. However, our model assumed that the number of nodes in the system is larger than C, the average number of nodes to be contacted for a partial lookup. When n is smaller than C, this assumption is not true, and partial lookups with few results behave more like total lookups. Consequently, it is possible to get away with a smaller domain size when n is very small.

E. Scalability w.r.t. data

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 0.6 messages/second to 5.5 messages/second (a near-linear cost increase); on the other hand, the overall cost for Adlib, with a domain size of 256, rises only to 0.86 messages/second.

3.3 Summary

We have seen that the use of Adlib can lead to a large improvement in overall system cost, compared to the use of a global index. In addition, Adlib scales well with respect to the number of tuples per node. We have observed that the optimal domain size, when optimizing for partial-lookup queries, is roughly constant and between 200 and 300 for a wide range of network sizes (for the query rates and node lifetimes observed in Gnutella). All the above conclusions are consistent with our model of Adlib which suggested that the optimal domain size is constant for partial lookups, and proportional to $\sqrt{n}$ for total lookups.

4 Designing a Self-tuning Adlib

Having seen that the ideal Adlib structure should have a number of domains that varies with network size, we now discuss how to develop intra- and inter-domain overlay networks, as well as mechanisms for data indexing and queries, to enable a dynamic, self-tuning index that approaches the optimal index configuration even as the number of nodes in the system varies over time.

A natural way of varying the number of domains used in Adlib is to split a domain into two whenever the number of domains is too few, and to merge two domains whenever there are too many domains. Performing such domain splitting (and merging) introduces multiple challenges:

The Overlay Problem: Nodes should not be required to abandon their existing overlay links and set up new links on a domain split, since such re-linking can prove very expensive. Query routing and broadcast must still operate efficiently under splits.

The Re-indexing Problem: The splitting or merging of domains requires a corresponding splitting and merging of indexes. Such re-indexing can prove expensive and must be made as efficient as possible.

The Atomic-Split Problem: Domain splits and merges must not require synchronization of all nodes in the domain, since such synchronization may prove impossible in a dynamic P2P system. Queries must continue to succeed even when only a fraction of the nodes have split, while others have not.

We will presently show how each of the above challenges may be handled.

Domains as a Binary Tree We can visualize the set of domains at any point of time as being at the leafs of a binary tree, as shown in Figure 6(a). Initially, all nodes are in a single "root" domain D. As the domain grows larger, it splits into two domains D_0 and D_1. Domain D_0 may itself grow larger over time, and split into D_{00} and D_{01} and so on, thus ensuring that the leafs of the binary tree correspond to the set of domains. Similarly, two sibling domains may merge into a single domain, which corresponds to deleting two sibling leaves in the tree. The *ID* of each domain is defined as the label along the path to the domain from the root; thus, D_{00} has ID 00, D_1 has ID 1, and so on.

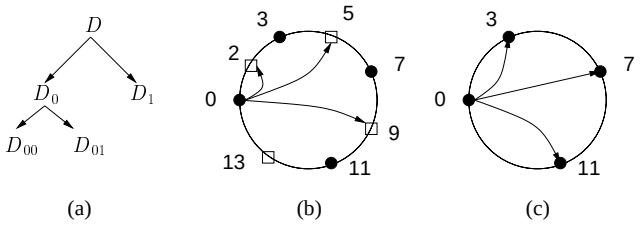

Figure 6. (a) A Hierarchy of Domains (b) Standard Chord links (c) Altering Chord to make it hierarchical

Assigning Nodes to Domains When a domain, say D, splits into D_0 and D_1, we need a mechanism to decide which nodes go to D_0 and which go to D_1. We assume that each node m chooses a random N-bit *hierarchy ID* $H(m)$ when it joins the system (for some large value of N). We can then identify what domain the node m belongs to: the leaf domain whose ID is a prefix of $H(m)$. Thus, in our example, all nodes with a prefix 0 go to D_0, and nodes with prefix 1 go to D_1. Note that $H(m)$ is completely *independent* of the intra-domain ID of node m that may determine its intra-domain links and index allocation.

4.1 The Overlay Problem

Let us first consider the challenges in constructing the intra-domain overlay network. Imagine all nodes are initially in domain D and have constructed an overlay network among themselves. As before, we assume the index is hash-partitioned across nodes, and Chord [18] is used as the overlay network. Figure 6(b) shows the set of links made by node 0 in a Chord [18] network. (The labels of nodes stand for their intra-domain ID.) At some point, domain D may split into D_0 and D_1, with each node going to D_0 or D_1 on the basis of its hierarchy ID. The figure depicts D_0 nodes by solid circles, and the D_1 nodes by hollow squares.

After the split, the D_0 nodes form an overlay network all by themselves (shown in Figure 6(c)), as do the D_1 nodes. In standard overlay networks such as Chord, the set of links for the D_0 and D_1 Chord networks is very different from

the set of links in the original network for D.[1] For example, node 0 is particularly unlucky, as it needs to establish three new links to nodes 3, 7 and 11 in the D_0 network. Thus, the split can prove to be very expensive.

Thus, our problem is the following: *Devise an overlay network that allows domains to split and merge without requiring any links to change at all, while still enabling routing in $O(\log n)$ hops with $O(\log n)$ links per node.* Our solution is to avoid using Chord and build a different overlay network instead. The key observation here is that the link structure for D should be set up *in anticipation* of the possibility that D may split into D_0 and D_1. If we could ensure that the network of nodes in D already *contains* the D_0 and D_1 networks as subgraphs, there would be no need to create new links when the domain splits. Of course, this process needs to happen recursively; since D_0 may itself split later on, the D_0 network should contain the D_{00} and D_{01} networks as subgraphs, and so on.

Exploiting this observation, we construct a network called Crescendo [7] with such recursive structure, while still ensuring $O(\log n)$ links per node and routing in $O(\log n)$ hops[2].

Summary of Crescendo We present a high-level summary of Crescendo, which is necessary to explain the rest of our algorithms. For more details, see [7]. The way a new node joins the system and sets up links is shown in Algorithm 1. The intuition behind the algorithm is simple: a node first joins and sets up links to other nodes at the lowest level in the hierarchy, and progressively adds a few more links at each higher level.

To illustrate, consider the set of nodes in Figure 6(b), and imagine node 0 is the last node being inserted. Let us suppose that there are only two levels in the hierarchy. Node 0 first joins at the "lowest level" domain D_0 and sets up links with some D_0 nodes, choosing links just as in Chord; the links it sets up are as in Figure 6(c). It then "goes up" a level to form additional links with other nodes in domain D. However, the only additional links it creates are to (some) D_1 nodes between itself and its successor in D_0 (node 3) – in this case, to node 2.

Crescendo, with its recursive structure, solves our problems since no new links need to be created on domain splits and merges, and still offers us all the desired intra-domain routing properties.

The Inter-Domain Structure Having fixed the intra-domain overlay problem, we now turn our attention to constructing an *inter-domain* network to enable iterative propagation of a query to any number of domains. Our solution is inspired by the hierarchical structuring of domains as a binary tree, and bears some resemblance to overlay networks such as Pastry [16]. Each node establishes a set of *inter-domain* links, one link at each level of the domain hierarchy, as defined below.

Algorithm 1 CrescendoJoin(Node m)

1: Let *Pre*(a, b) = length of common prefix of $H(a)$ and $H(b)$.
2: **for** $l = length(H(m))$ downto 0 **do**
3: $d_{min} = \min_{m'}\{Dist(m, m')|Pre(m, m') = l + 1\}$
4: $D_l = \{m'|$*Pre*$(m, m') = l$ and $Dist(m, m') < d_{min}\}$
5: Set up links to (a subset of) nodes in D_l, as dictated by the *Chord rule.*
6: **end for**

INTER-DOMAIN LINKS: *Consider a node m with hierarchy ID $b_1b_2b_3 \ldots b_N$, where each b_i is either zero or one. Let S_i denote the set of nodes with hierarchy ID prefix $b_1b_2 \ldots b_{i-1}\overline{b_i}$. Node m establishes a set of N inter-domain links, with the i^{th} link being to the closest predecessor of m in the set S_i. (Here, predecessor is defined on the value of nodes' intra-domain IDs.)* Note that the set S_i will be empty for large values of i, thus limiting the number of actual links established.

To explain the above definition, a node m forms inter-domain links to some node in the sibling subtree at each level of the hierarchy. The link is not just to any node in the subtree, but to the node whose intra-domain ID is closest in value to m's intra-domain ID while being less than or equal to it. To illustrate, node 0 in our example forms a level-1 link to its closest predecessor in domain D_1, i.e., node 13. It forms a level-2 link to its closest predecessor in D_{01}, and so on.

The total number of inter-domain links per node is at most the number of non-empty levels in the domain hierarchy; when hierarchy IDs are chosen randomly, the number of such levels is $O(\log n)$, ensuring that each node maintains only $O(\log n)$ inter-domain links. The details of how to maintain this structure as nodes join and leave are straightforward. Since links are set up at all levels of the hierarchy ahead of time, the splitting of a domain into two does not necessitate the formation of any additional links to maintain the structure.

Queries and Iterative Broadcast For now, imagine that domain splits and data re-indexing occur instantaneously, so that each domain indexes its local content at all times. We describe query routing under this assumption, and relax this assumption in Section 4.3.

Partial Lookups Let us first consider partial-lookup queries. A query is first evaluated in its own domain, by routing it to the relevant index node using the links defined in Algorithm 1. If the answers within the domain are insufficient, the query is *iteratively* broadcast to more and more domains, roughly doubling the number of domain in each

[1] In expectation, half the links in D will exist in D_0 or D_1, while the other half are new links.

[2] Another network with such recursive structure is the skip graph [4]. We use Crescendo as it is more general than the skip graph, as we discuss in Section 5.

step. We may visualize this process in terms of the domain hierarchy: a query is first sent to the sibling domain in the hierarchy at level, say, l, then to all domains in the sibling subtree at level $l-1$, then to the sibling subtree at level $l-2$, and so on, until all domains are reached. (See pseudocode in Algorithm 2.)

To illustrate, consider a query initiated in domain D_{00} whose relevant index is stored at node 0. If answers at 0 are insufficient, node 0 sends the query to domain D_{01}, using its level-2 inter-domain link to its predecessor in D_{01}, say p. Once the query reaches node p, it is routed within D_{01} using intra-domain links to reach the relevant node in that domain. Note, however, that since p is the immediate predecessor of 0, it is very likely that p, or one of its immediate successors, stores the relevant index entry for domain D_{01}. Therefore, the query can be answered in domain D_{01} using just a constant number of messages in expectation.

If the answers from domain D_{01} are still insufficient, node 0 sends the query up to the next level in the hierarchy: all leaf domains under D_1. Node 0 uses its level-1 inter-domain link, i.e., to node 13, in order to propagate the query to D_1. If D_1 is a leaf domain, node 13 would simply route the query within that domain to obtain results. But D_1 may not be a leaf domain itself, and may have many leaf domains in its sub-tree. In this case, node 13 recursively broadcasts the query to all domains in the D_1 subtree, with all answers being sent to node 0. The algorithm for this recursive broadcast is almost identical to the total-lookup algorithm that we describe next. (See pseudocode in Algorithm 3.)

Total Lookups When a query requires all answers in the system, it needs to be broadcast to all domains. The first step in a total lookup is the same as the partial lookup: the query is first routed within the original domain to get to the responsible node, node 0 in our example. Node 0 then sends the query using *all* "inter-domain" links that connect to nodes outside its current domain. In our example, node 0 sends the query to node 13 in D_1, designating node 13 responsible for broadcasting the query to all domains in the entire D_1 subtree. Node 0 also sends the query to its predecessor in D_{01} designating it responsible for broadcasting to all domains in the D_{01} subtree. (Of course, in our example, D_{01} is a domain itself.) Although node 0 may also have more "inter-domain" links to other nodes in D_{00}, it does not use them for broadcast, since it realizes D_{00} is a domain by itself. (See pseudocode in Algorithm 4.)

The primary difference of this approach from the partial-lookup algorithm is that node 0 uses all its inter-domain links simultaneously, instead of using them one at a time and waiting for results. Note that the broadcast algorithm requires the ability to broadcast within a subtree as a subroutine. This is easy to do: a node required to broadcast in a subtree at level i uses all its inter-domain links except the links for levels 1 to $i-1$.

Algorithm 2 PartialLookupQuery(StartNode s, Query q)

```
1: Answers=FindIntraDomainAnswers(s, q)
2: Let the query reach node m within the domain. Let the
   current domain level be l.
3: Let m's inter-domain link at level x be to node n_x.
4: for x = l downto 0 do
5:    If sufficient answers, break.
6:    Answers=Answers+PropagateQuery(n_x, x+1, q,m)
7: end for
8: return Answers to node s;
```

Algorithm 3 PropagateQuery(CurrentNode m, LevelOfPropagation h, Query q, SourceNode s)

```
1: Let the current domain level for node m=l
2: Let m's inter-domain link at level x be to node n_x.
3: for x = h to l - 1 do
4:    PropagateQuery(n_x, x + 1, q, s)
5: end for
6: FindIntraDomainAnswers(n, q) and return them to s
```

Algorithm 4 TotalLookupQuery(StartNode s, Query q)

```
1: Route the query within the domain; let it reach node m.
2: PropagateQuery(m,0,q,s)
```

We present analysis of the partial- and total-lookup algorithms in Appendix B. We simply note here that the message complexity and the latency experienced by queries are as described in Theorem 1 (Sec. 2.2).

Summary We have now described intra-domain and inter-domain overlay networks in which all nodes have $O(\log n)$ links each, which allows efficient routing and broadcast of queries within and across domains respectively. Domains may split and merge at will, and do not require changes to the link structure.

4.2 The Re-indexing Problem

Having taken care of the overlay problem, we now turn to the re-indexing problem. When domain D splits into D_0 and D_1, index entries need to be moved since the D_0 nodes should now index only content in D_0, while D_1 nodes need to index D_1 content. It turns out that such re-indexing is not too expensive, since only pairs of nodes need to communicate to exchange index entries.

Consider the example network in Figure 6(b). Initially, all nodes are in the same domain, so that node 0 indexes *all* content hashing to the range $[0, 2)$, while node 2 indexes all content in range $[2, 3)$. When the domain splits, node 0 should index only D_0 content in range $[0, 3)$, while node 2 indexes D_1 content in $[2, 5)$. In order for this transformation to occur, node 0 needs to obtain index entries for D_0 content in $[2, 3)$, which it may receive by communicating with node

2. (Recall that the two nodes already have a link between them in the network.) Similarly, node 2 communicates with node 3 to obtain all its index entries. In general, each D_0 (resp., D_1) node only needs to receive data from one D_1 (resp., D_0) successor (in expectation) in order for the re-indexing to complete. Such data exchange is made easier by the fact that these pairs of nodes also have a link to each other in the interconnection network.

4.3 The Atomic-Split Problem

Two questions remain. (1) When should a domain be split into two, and when should two domains merge? (2) How can the split be performed without requiring all nodes to act simultaneously? How do we handle queries that are issued while a split is going on?

When to Split To answer the first question, we observe that the optimal domain size is a function of the number of nodes in the network when optimizing for total lookups, and is a fixed constant when optimizing for partial lookups (assuming that other system characteristics such as average node lifetime are stable). If a node can estimate the total size of the network, as well as the current size of its domain, it can decide whether the current domain size is too large or too small. If the domain size is too large (say, greater than twice the optimal size), the node may initiate a domain split operation. If the size is too small (say, less than half the optimal size), the node initiates a merge operation.

Network-size estimation is a well-studied problem and standard techniques, e.g., [13], can be used to estimate both the network size and the domain size. Since domain sizes are fairly small, one may even obtain more accurate estimates using a broadcast message to count the number of nodes in the domain. Hereonin, we simply assume that it is possible to obtain reasonable estimates of domain size and network size. (Note that when optimizing for partial lookups, we do not need an estimate of the network size. We merely have to check if the domain size is much greater than or much less than a constant.)

Splitting without synchronization For the second problem of how all nodes in a domain split at the same time, we simply allow each node to figure out for itself when the domain ought to split, and then act upon that decision. Its actions consist of contacting nodes in the new sibling domain and obtaining index entries from them for content in its own new domain, and are described in Algorithm 5.

Observe that the algorithm is completely independent of whether other nodes have chosen to split or not. Therefore, it is entirely possible, and indeed inevitable, that a situation arises where some nodes in the domain have chosen to split, and have modified their index content appropriately, while other nodes are yet to split. We note, however, that such a situation will not last forever, since all nodes will eventually come to the same conclusion on whether to split or not, based on the domain size estimate.

Algorithm 5 DomainSplit(Node m)

```
1: loop
2:    l=Current domain level of m
3:    n̂=Estimate of current domain size
4:    if n̂ > Split Threshold then
5:       s = Successor of m at level l + 1
6:       for all m' between m and s s.t. Pre(m, m') = l do
7:          {There is only one m' in expectation}
8:          Get index entries for level-(l + 1) domain from m'
9:       end for
10:   end if
11: end loop
```

But what happens to queries that are executed in the system during this intermediate stage where some nodes have split and others have not? The careful establishment of the intra-domain and inter-domain links ensures that queries are completely unaffected by the asynchronous splitting of domains. Thanks to an intra-domain routing property called *path convergence* [7], we can prove that all queries will continue to retrieve all the relevant answers. We now explain this path convergence property, and show why it solves the atomic-split problem.

Path Convergence *Let S be a set of nodes with a common l-bit prefix in their hierarchy ID. Consider a node m not in S that shares a prefix of length $l' < l$ with all nodes in S. Then, the routing path from every node in S to node m, using intra-domain links, must go through some common node c in S. This common node c is the closest predecessor of node m in S.*

To illustrate path convergence, consider some node m in D_0 attempting to route a message to node 2 (treating the entire system as a single domain). Path convergence states that, no matter what m is, the routing path from m to node 2 *must go through node* 0. (Note that 0 is the closest predecessor of 2 in domain D_0.)

To understand why path convergence helps in ensuring correct queries, let us consider a query for a key K that hashes to the value 2. Index entries for K could potentially be stored at a number of nodes. If node 2 believes that its domain consists of the entire system D, 2 would hold all entries for K. If node 2 believes its domain is D_1, it would have only the index entries available in D_1, and would have handed off the entries for D_0 data to node 0. Of course, node 0 might itself believe its domain to be D_{00}, in which case node 0 would have handed off some entries to the relevant node in D_{01}, and so on. In any case, the relevant index entries for key K is stored over *a set of nodes distributed across all branches of the domain hierarchy*. (Also observe that the nodes in this set are connected by inter-domain links.)

Now consider a node m, say in D_{00}, that initiates a query

for key K. Since nodes don't synchronize when performing splits, node m does *not* have any idea which set of nodes contain the index entries for K. In order to answer the query, node m uses its intra-domain links and routes the query towards K's hash value, i.e., 2, *treating the entire system as a single domain*.

By the path convergence property, this query message *must reach node* 0 *along the way*. If node 0 already has performed domain splits and is responsible for storing some entries for key K, node 0 intercepts the query message, terminates intra-domain routing, and returns the answers it contains to m; more answers, if necessary, can be discovered by node 0 using its inter-domain links to reach domains D_{01}, D_1, etc. If node 0 still believes that its domain is the entire system D (i.e., node 2 still contains all entries for key K), it does not intercept the query message, and continues to route it using intra-domain links. The query would eventually reach node 2, which contains all the answers for K in this case and can return these answers to m. By a similar argument, it is easy to show that (a) all queries are able to retrieve all the data in the system in an appropriate fashion, and (b) update broadcast messages can also be transmitted to ensure that the relevant nodes will receive a broadcast message informing them of node departures.

4.4 Discussion

Domain Merges So far, we have discussed only the splitting of domains. Domain merges are the exact inverse of a domain split. Since no links were ever deleted when a domain split, no new links need to be created when a domain that split earlier becomes a single domain again. Query routing during a domain merge is identical to that during a split, and is automatically taken care of, as discussed earlier. Note that there is *hysteresis* to ensure that domains don't split and merge all the time. A domain splits when its size becomes a constant factor larger than the optimal, while two domains merge if their cumulative size is a constant factor less than the optimal.

The Cost of Split/Merge The primary cost associated with a domain split/merge is that of transferring index entries between neighboring nodes. Although this cost is dependent on the size of the indexes stored at each node, we observe that the cost isn't too high because splits and merges happen infrequently: a domain split occurs only when the total number of nodes in the system doubles, while a merge occurs only when the total number of nodes halves. Although P2P systems are dynamic in terms of node arrivals and departures, the *size* of the P2P system changes more slowly[3].

A secondary cost is that of nodes estimating domain sizes in order to determine when to split or merge. This cost is typically very small and the estimation can often be done by piggybacking on other traffic in the P2P system [13].

[3]Typically, system size changes by a large factor about once a day [17], corresponding to nodes joining during the day and leaving late at night

4.5 Evaluation

We now evaluate how Adlib, with dynamic variations in the number of domains, scales as the number of nodes in the system increases continuously. Our objective is to measure two things: (a) the overall system cost, measured as the sum of the query and index-maintenance costs, for different network sizes, as the network evolves, and (b) the cost of the adaptation itself, i.e., the overhead involved in performing splits and merges themselves.

Our experiment begins with $n = 4096$ nodes in the system, and the size of the system increases as nodes progressively join. As the number of nodes increases, domain splits may occur as dictated by the splitting algorithm. We optimize for partial-lookup queries and, in keeping with the evaluation in Section 3.2, require that the maximum domain size be 512, i.e., a domain is expected to split when it becomes any larger. We assume that nodes have an accurate estimate of domain size; relaxing this assumption only serves to stagger a domain split over a slightly larger period of time, and has little impact on costs.

The continuous growth in system size is interspersed with periods of "stability"; during these periods of stability, nodes join and leave at an equal rate (ensuring that the system size remains stable) and issue queries. Node lifetimes as well as query rates are drawn from Gnutella measurements, just as in Section 3.2. Once enough time passes in the stable state to obtain a good estimate of query and update costs at that network size, the system once again switches into a "growth" phase with nodes joining the system continuously until the next "stable" phase. Note that no domain splits and merges occur in the stable phases as the balancing of node joins and leaves ensures that domain sizes do not change much.

Figure 7 plots the overall system cost, measured during the stable phases at the different system sizes observed during the evolution of the system. The solid curve shows the cost incurred in the system, with each marked point on the curve reflecting measurements during a stable phase at which the cost is measured. The cost shows a step-like behaviour, dropping every time domain splits occur at network sizes close to a power of 2. The dashed curve underneath in the figure shows the *optimal* achievable cost for the corresponding network sizes, using the *best* choice of domain sizes that we described in Section 3.2. (Note that the y-axis does not start at zero.) We see that our dynamic adaptation of domain size is fairly close in performance to the optimal achievable cost for all network sizes.

Split Overhead To quantify the overhead of *performing domain splits* themselves, we need to know the rate at which

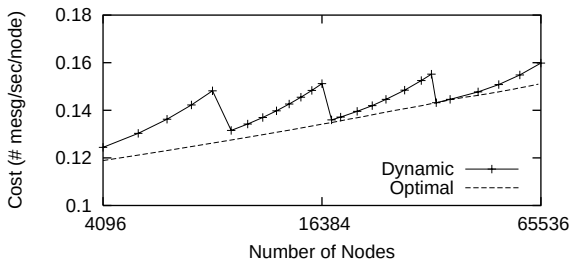

Figure 7. Overall System Size as a function of system size

the network grows in size. Say the network doubles in size every D seconds. The overhead of performing *one* domain split is simply the cost of re-indexing data. For re-indexing, pairs of nodes exchange portions of the index; if each node has t tuples on average, the number of index entries transferred out per node is $t/2$. Since all the entries need to be sent to only one other node in expectation, and assuming each index entry is i bytes long, the bandwidth cost of performing such a split, averaged over the time for the system to double in size, is $it/2D$ bytes/sec.

In our simulation, $t = 307$, and we assume an index entry is 10 bytes long; A typical Gnutella system doubles or halves in size about once a day [17]. Even assuming conservatively that our system doubles in size every hour, the bandwidth overhead of splits per node is under 1 byte/second and is thus very small. Even when t is much larger, say $t = 10^6$, the overhead is under 100 bytes/second when the system doubles/halves in size daily.

5 Adapting to the Physical Network

So far, we have ignored the fact that different nodes may be in physically disparate locations, spread across a wide-area network. Ideally, we would like to ensure all communication on the overlay-network actually occurs between physically nearby nodes, to ensure low query latency and reduce bandwidth usage. Adlib allows adaptation to such physical-network proximity in a natural fashion.

The Adlib structure continues to operate efficiently even when nodes do not choose their hierarchy ID at random. Thus, nodes could select their hierarchy ID in such a fashion that physically close nodes also possess hierarchy IDs with a common prefix. (Note that, after a certain fixed-length prefix, the remaining bits of the hierarchy IDs should continue to be chosen at random, in order to avoid extreme cases of all nodes choosing exactly the same ID.)

We do not discuss exactly how nodes assign themselves such a hierarchy ID, 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 could measure its physical-network distance to these beacons, and chooses a hierarchical ID based on its proximity to each of the beacons, in a fashion similar to that described in [15].

Such an arrangement has the effect of ensuring that nodes within a domain are physically close to each other. Consequently, index construction within a domain becomes optimized for the physical network, as content is indexed only at physically nearby nodes. Query routing within the domain is also efficient. Moreover, for partial-lookup queries that desire only a limited number of results, the query routing algorithms first provide results that are physically nearby in the network, and slowly identify results that are further and further away, until enough answers are obtained.

Such a query execution strategy is not only efficient in terms of network bandwidth usage and query latency, but also offers some *nearest-k* semantics, i.e., results that are returned are that obtained from physically close nodes which can be useful from the application perspective.

6 Related Work

Distributed Hash Tables [14, 18, 16, 10], such as Chord, have been proposed as a substrate for a variety of distributed applications. The PIER project [11] uses DHTs specifically for the problem of building distributed indexes over relational data to enable single and multi-table queries. This work is complementary to ours in that it enables more complex queries on top of basic index structures, while our work focuses on the most efficient way to construct the basic index.

P-Grid [3] uses indexes to support queries over ordered and unordered data. P-Grid also investigates the issue of index updates, but its focus is on the update of individual tuple values, rather than on the insertion and removal of tuples themselves, with dynamic node joins and leaves.

In file-sharing systems such as Gnutella [1], each node indexed its own content, and queries were flooded across nodes. KaZaa [2] extended the Gnutella approach with the use of *supernodes* which serve as a proxy for nodes with lower bandwidth. Adlib also extends to supporting the use of such supernodes. Both Gnutella and KaZaa suffer from high query-execution cost and poor query recall. In comparison, the Adlib index reduces the overall system cost by more than an order of magnitude.

Hybrid solutions such as YAPPERS [8] 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.

7 Conclusions

We have introduced a self-tuning index structure that trades off index-maintenance cost against the benefit for queries. We have shown that Adlib can reduce the system overhead by a factor of four, compared to a global index. We have shown how to devise overlay networks, as well as indexing and querying mechanisms to support the dynamic and self-tuning index without the need for any global synchronization for index re-organization. The performance advantages of Adlib are amplified when the system is optimized for the underlying physical network, as Adlib enables natural support for reducing update costs of the index, and finds partial-lookup results in a network-efficient fashion.

References

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

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

[3] K. Aberer. P-grid: A self-organizing access structure for P2P information systems. In *Proc. CoopIS*, 2001.

[4] J. Aspnes and G. Shah. Skip graphs. In *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 2003.

[5] 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.

[6] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to p2p systems. In *Proc. VLDB*, 2004.

[7] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in g major: Designing DHTs with hierarchical structure. In *Proc. ICDCS*, 2004.

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

[9] P. Ganesan, Q. Sun, and H. Garcia-Molina. Adlib: A self-tuning index for dynamic p2p systems. Technical report, Stanford University, 2004.

[10] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. 14th ACM SPAA*, 2002.

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

[12] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th ACM Symposium on Theory of Computing (STOC 1997)*, pages 654–663, 1997.

[13] G. S. Manku. Routing networks for distributed hash tables. In *Proc. PODC*, 2003.

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

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

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

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

[18] 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.

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

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

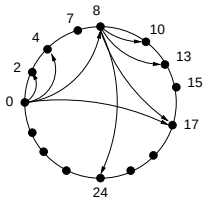

Figure 8. An Example Chord Network

A Chord and Broadcast on Chord

Chord 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 8 depicts an example of nodes in a 5-bit space. The *distance* from node m to node m', $Dist(m, m')$ 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 $0 \leq i < N$. We call this the *Chord rule*. 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.

Chord enables efficient routing of a message between any two nodes by greedy, clockwise routing. For example, node 0 would route to node 15, by sending the message to node 8, which forwards it to node 13, and from there to node 15. This mechanism requires only $O(\log n)$ messages for routing between any arbitrary pair of nodes. Moreover, when nodes join or leave the system, only $O(\log n)$ messages to restructure the overlay network appropriately for the new set of nodes.

Chord was 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. 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 the Chord overlay network was proposed for routing, an obvious extension also allows efficient *broadcast* of messages from one node to all other nodes. A node initiating the broadcast 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.

It is equally straightforward to achieve broadcast on other overlay networks besides Chord. In fact, the same broadcast algorithm may be used for broadcast on Crescendo, the overlay network used by Adlib in its intra-domain structure.

B Proofs

Theorem 1. *If there are n nodes in the system distributed uniformly across k domains, the following statements hold with high probability:*

(a) the total out-degree of each node is $O(\log n)$,

(b) the number of messages exchanged to handle a node join or leave is $O(\log n)$,

(c) a total-lookup query initiated by any node takes $O(k + \log(n/k))$ messages to obtain all answers,

(d) a partial-lookup query that contacts f domains takes $O(f + \log(n/k)$ messages to obtain answers,

(e) if the latency of a message transmission is one time unit, a total-lookup query is answered in $O(\log n)$ time units, and a partial-lookup query in $O(\log^2 k + \log(n/k))$ time units.

Proof. (a) The number of intra-domain links per node is $O(\log n)$ as shown in [7]. Now, consider the number of inter-domain links set up by a node m according to the definition in Section 4.1. Node m sets up one inter-domain link for each value of $i, 0 < i \leq N$, that the set S_i is non-empty. Since the set S_i consists of all nodes with a specific i-bit prefix in their hierarchy ID, and nodes choose their hierarchy ID independently and at random, it is easy to see that, with high probability, S_i is empty for all $i > i'$ for some $i' = O(\log n)$. Consequently, the total number of links per node is $O(\log n)$ with high probability.

(b) The number of messages to fix the intra-domain network after a node join/leave is $O(\log n)$ [7]. For the inter-domain network, observe that only $O(\log n)$ other nodes are affected by a node join/leave, by part (a). Each of the affected nodes can re-connect to the appropriate node using only a constant number of messages with high probability, just as in the case of the intra-domain network.

(c) The number of intra-domain messages to find query answers in the source domain is $O(\log(n/k))$ [18, 7]. The number of additional messages necessary to forward the query to all the domains is $k - 1$, by the use of inter-domain broadcast. Within each domain, the number of nodes be-

tween the first recipient of the query, and the query destination is a constant with high probability, thus requiring only $O(1)$ messages within each domain.

(d) The argument is identical to (c) above, noting that the iterative broadcast terminates after the query reaches only $O(f)$ domains with high probability.

(e) The latency of an intra-domain lookup is $O(\log(n/k))$ [18, 7]. The latency of broadcasting the query to all k domains is $O(\log k)$, since it is bounded by the depth of the lowest-level domain in the domain hierarchy. The latency of query routing within each domain is $O(1)$, by the argument in part (c). Thus, the latency of a total-lookup query is $O(\log n)$. For partial lookups, the latency of broadcast is at most $O(\log^2 k)$, since broadcast is iterative with both the number of iterations and the length of each iteration being $O(\log k)$, leading to the desired bound on overall latency.

□

Theorem 2. *(a) The cost of index creation and maintenance with update broadcast is* $(3t/L)(1 - k/n) + n/(Lk)$ *messages per node per second.*

(b) For a time-out period T_o*, the cost of index creation and maintenance with time-outs is* $(3t/L)(1-k/n)+\frac{n}{kT_o}(1-(1-k/n)^t)$ *messages per node per second.*

Proof. (a) Consider the costs of index creation and maintenance associated with a particular node m. When m joins the system, its content needs to be inserted into the index. The probability that node m itself is responsible for a particular tuple is k/n. Additionally, index entries need to be migrated from the neighbor of m to itself; when all nodes have t tuples each, the number of migrated tuples is $t(1 - k/n)$. When m leaves, the index tuples that m stored have to be re-inserted into the system; there are $t(1 - k/n)$ such tuples. Finally, the information about m's departure has to be broadcast to the rest of the nodes in the system, which requires n/k messages. The above costs are incurred over the lifetime L of node m, leading to the desired expression for costs[4]

(b) With the time-out mechanism, the costs of index insertion on node join, index migration on node join, and index re-insertion on a node leave are the same as in part (a). However, there is the additional cost of periodic keep-alive messages. If a node has t tuples, the expected number of distinct nodes that have an index entry for at least one of those t tuples is $(1 - (1 - k/n)^t)n/k$. We assume that one keepalive message needs to be sent to each of these distinct nodes every T_o time units, leading to the desired result.

□

[4]We have used an approximation to simplify the expression here; the initial insertion of content requires nearly $0.5\log(n/k))$ messages per tuple, while subsequent index modifications require only one message per tuple. We ignore this minor difference, both for time-outs and update broadcast.