ISSN (ONLINE): 2395-695X
ISSN (PRINT): 2395-695X
International Journal of Advanced Research in Basic Engineering Sciences and Technology (IJARBEST)
Volume 2, Special Issue 19, October 2016

An Index based over Distributed Hash Tables for Peer to Peer Systems

Mrs. D. Kavitha
P.hD. Research Scholar
Nandha Arts and Science College,
Koorapalyam Pirivu, Perundurai Road, Erode-52.
E-Mail : rkjaidanya46@gmail.com
Cell :9095622299

Dr. R. Ramkumar
Head / Computer Applications
Nandha Arts and Science College,
Koorapalyam Pirivu, Perundurai Road, Erode-52.
E-Mail : ramkumar2006@gmail.com
Cell :9095910555

***Abstract*—**Distributed Hash Tables are scalable, robust, and self-organizing peer-to-peer systems that support exact match lookups. This paper describes the design and implementation of a Pre x Hash Tree - a distributed data structure that enables more sophisticated queries over a DHT. The Pre x Hash Tree uses the lookup interface of a DHT to construct a trie based structure that is both efficient (updates are doubly logarithmic in the size of the domain being indexed), and resilient (the failure of any given node in the Pre x Hash Tree does not affect the availability of data stored at other nodes).

Categories and Subject Descriptors
Comp. Communication Networks
Data Structures
Information Storage and Retrieval

General Terms
Algorithms, Design, Performance

Keywords
Distributed hash tables, data structures, range queries

1. INTRODUCTION

The explosive growth but primitive design of peer-to-peer le-sharing applications such as Gnutella [7] and KaZaa [29] inspired the research community to invent **Distributed Hash Tables** (DHTs) [31, 24, 14, 26, 22, 23]. Using a structured overlay network, DHTs map a given key to the node in the network holding the object associated with that key; this lookup operation lookup(key) can be used to sup-port the canonical put(key, value) and get(key) hash table operations. The broad applicability of this lookup interface has allowed a wide variety of system to be built on top DHTs, including le systems [9, 27], indirection services [30], event notification [6], content distribution networks [10] and many others

DHTs were designed in the Internet style: scalability and ease of deployment triumph over strict semantics. In particular, DHTs are self-organizing, requiring no centralized authority or manual configuration. They are robust against node failures and easily accommodate new nodes. Most importantly, they are scalable in the sense that both latency (in terms of the number of hops per lookup) and the local state required typically grow logarithmically in the number of nodes; this is crucial since many of the envisioned scenarios for DHTs involve extremely large systems (such as P2P mu-

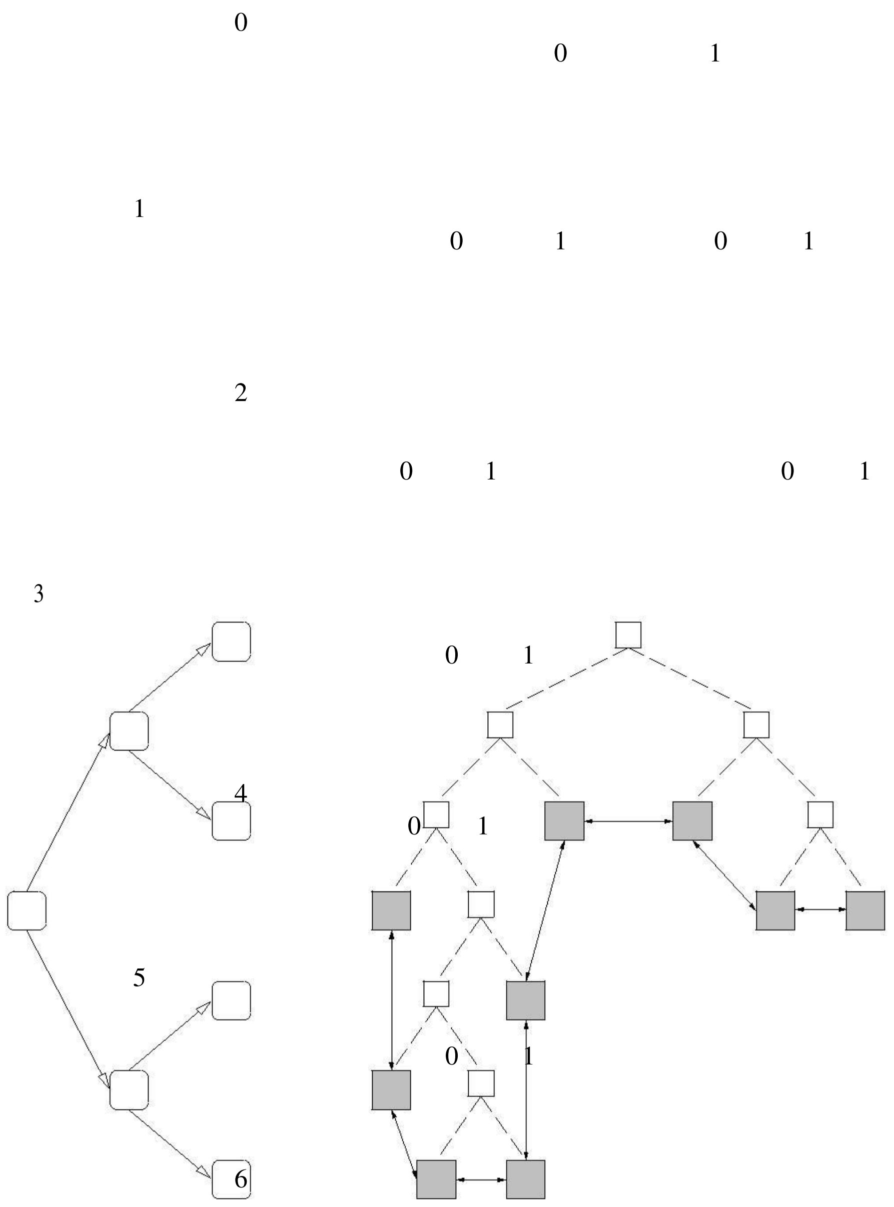
this lookup interface has allowed a wide variety of system to be built on top DHTs, including le systems [9, 27], indirection services [30], event notification [6], content distribution networks [10] and many others.

DHTs were designed in the Internet style: scalability and ease of deployment triumph over strict semantics. In particular, DHTs are self-organizing, requiring no centralized authority or manual configuration. They are robust against node failures and easily accommodate new nodes. Most importantly, they are scalable in the sense that both latency (in terms of the number of hops per lookup) and the local state required typically grow logarithmically in the number of nodes; this is crucial since many of the envisioned scenarios for DHTs involve extremely large systems (such as P2P mu-

sic le sharing). However, DHTs, like the Internet, deliver "best-e ort" semantics; put's and get's are likely to succeed, but the system provides no guarantees. As observed by others [36, 5], this conflict between scalability and strict semantics appears to be inevitable and, for many large-scale Inter-net systems, the former is deemed more important than the latter.

While DHTs have enjoyed some success as a building block for Internet-scale applications, they are seriously de cient in one regard: they only directly support **exact match** queries. Keyword queries can be derived from these exact match queries in a straightforward but ine cient manner; see [25, 20] for applications of this to DHTs. Equality joins can also be supported within a DHT framework; see [15]. However, **range queries**, asking for all ob-jects with values in a certain range, are particularly di cult to implement in DHTs. This is because DHTs use hashing to distribute keys uniformly and so can't rely on any structural properties of the key space, such as an ordering among keys.

Range queries arise quite naturally in a number of



Leaf nodes	**Keys**
000*	000001
	000100
	000100
00100*	001001
001010*	001010
	001010
	001010
001011*	001011
	001011
0011*	
01*	010000
	010101
10*	100010
	101011
	101111
110*	110000
	110010
	110011
	110110
111*	111000
	111010

Figure 1: Pre x Hash Tree

potential application domains:

Databases Peer-to-peer databases [15] need to sup-port SQL-type relational queries in a distributed fashion. Range predicates are a key component in SQL.

Distributed computing Resource discovery requires locating resources within certain size ranges in a decentralized manner.

Location-aware computing Many applications want to locate nearby resources (computing, human orcommercial) based on a user's current location, which is essentially a 2-dimensional range query based on geographic coordinates.

Scienti c computing Parallel N-body computations [34] require 3-dimensional range queries for accu-rate approximations.

In this paper, we address the problem of e ciently supporting 1-dimensional range queries over a DHT. Our main contribution is a novel trie-based dis-tributed data structure called **Pre x Hash Tree** (hence-
forth abbreviated as PHT) that supports such queries. As a corollary, the PHT can also support heap queries (\what is the maximum/minimum ?"), prox-
imity queries (\what is the nearest element to X
?"), and, in a limited way, multi-dimensional ana-logues of the above, thereby greatly expanding the querying facilities of DHTs. PHT is e cient, in that updates are doubly logarithmic in the size of the domain being indexed. Moreover, PHT is self-organizing and load-balanced. PHT also toler-ates failures well; while it cannot by itself protect
against data loss when nodes go down[1], the failure of any given node in the Pre x Hash Tree does not a ect the availability of data stored at other nodes.

But perhaps the most crucial property of PHT is that it is built entirely on top of the lookup inter-face, and thus can run over any DHT. That is, PHT uses only the lookup(key) operation common to
all DHTs and does not, as in SkipGraph [1] and other such approaches, assume knowledge of nor require changes to the DHT topology or routing behavior. While designs that rely on such lower-layer knowledge and modi cations are appropriate for contexts where the DHT is expressly deployed for the purpose of supporting range queries, we ad-dress the case where one must use a pre-existing DHT. This is particularly important if one wants to make use of publicly available DHT services, such as OpenHash [18].

The remainder of the paper is organized as fol-lows. Section 2 describes the design of the PHT data structure. Section 3 presents the results of an experimental evaluation. Section 4 surveys related work and section 5 concludes.

2. DATA STRUCTURE

This section describes the PHT data structure, along with related algorithms.

2.1 PHT Description

For the sake of simplicity, it is assumed that the do-main being indexed is f0; 1g^{D} , i.e., binary strings

[1]But PHT can take advantage of any replication or other data-preserving technique employed by a DHT. of length D, although the discussion extends nat-urally to other domains. Therefore, the data set indexed by the PHT consists of some number N of D-bit binary keys.

In essence, the PHT data structure is a binary trie built over the data set. Each node of the trie is labeled with a pre x that is de ned recursively: given a node with label l, its left and right child nodes are labeled l0 and l1 respectively. The root is labeled with the attribute being indexed, and downstream nodes are labeled as above.

The following properties are invariant in a PHT.

1. (**Universal pre x**) Each node has either 0 or 2 children.

2. (**Key storage**) A key K is stored at a leaf node whose label is a pre x of K .

3. (**Split**) Each leaf node stores atmost B keys.

4. (**Merge**) Each internal node contains atleast (B + 1) keys in its sub-tree.

5. (**Threaded leaves**) Each leaf node maintains a pointer to the leaf nodes on its immediate left and
and immediate right respectively.[2]

Property 1 guarantees that the leaf nodes of the PHT form a **universal pre x set** [3]. Consequently, given any key K 2 f0; 1g^{D} , there is exactly one leaf node leaf (K) whose label is a pre x of K . Prop-
erty 2 states that the key K is stored at leaf (K). Figure 1 provides an example of a PHT contain-ing N = 20 6-bit keys with B = 4. The table on the right in Figure 1 lists the 20 keys and the leaf nodes they are stored in.

Properties 3 and 4 govern how the PHT adapts to the distribution of keys in the data set. Fol-lowing the insertion of a new key, the number of keys stored at a leaf node may exceed the threshold B, causing property 3 to be violated. To restore the invariant, the node splits into two child nodes, and its keys are redistributed among the children according to property 2. Conversely, following the deletion of an existing key, the number of keys con-tained in a sub-tree may fall below (B +1), causing property 4 to be violated. To restore the invari-ant, the entire sub-tree is merged into a single leaf node, where all the keys are aggregated. Notice the shape of the PHT depends on the distribution of keys; it is "deep" in regions of the domain which are densely populated, and conversely, "shallow"

[2] A pointer here would be the pre xes of neighboring leaves and, as a performance optimization, the cached IP address of their corresponding DHT nodes.

[3] A set of pre xes is a universal pre x set if and only if for every in nite binary sequence b, there is exactly one element in the set that is a pre x of b. in regions of the domain which are sparsely popu-lated. Finally, property 5 ensures that the leaves of the PHT form a doubly linked list, which en-ables sequential traversal of the leaves for answer-ing range queries.

As described this far, the PHT structure is a fairly routine binary trie. The novelty of PHT lies in how this logical trie is **distributed** among the peers in the network; i.e., in how PHT vertices are as-signed to DHT nodes. This is achieved by **hashing** the pre x labels of PHT nodes over the DHT iden-ti er space. A node with label l is thus assigned [4] to the peer to which l is mapped by the DHT, i.e., the peer whose identi er is closest to HASH(l). This hash-based assignment implies that given a la-bel, it is possible to locate its corresponding PHT node via a single DHT lookup. This \direct access" property is unlike the successive link traversals as-sociated with typical data structures and results in the PHT having several desirable features that are discussed subsequently.

2.2 PHT Operations

This section describes algorithms for PHT opera-tions.

2.2.1 Lookup

Given a key K , a PHT lookup operation returns the unique leaf node leaf (K) whose label is a pre x of K . Because there are (D + 1) distinct pre xes of K , there are (D + 1) potential candidates; an obvious algorithm is to perform a linear scan of these (D + 1) nodes until the required leaf node is reached. This is similar to a top-down traversal of the trie except that a DHT lookup is used to locate a PHT node given its pre x label. Pseudocode for this algorithm is given below.

```
Algorithm: PHT-LOOKUP-LINEAR
input  : A key K
output: leaf (K )

for i      0 to D do
    /*P_i (K ) denotes prefix of K of length i */
    node      DHT-LOOKUP(P_i (K ));
    if (node is a leaf node) then  return node ;
end
return f ailure;
```

How can this be improved ? Given a key K , the above algorithm tries di erent pre x lengths until the required leaf node is reached. Clearly, linear search can be replaced by **binary search** on pre x lengths. If the current pre x is an internal node of the PHT, the search tries longer pre xes. Al-ternatively, if the current pre x is not an internal node of the PHT, the search tries shorter pre xes. The search terminates when the required leaf node is reached. The decision tree to the left in Fig-ure 1 illustrates the binary search. For example. consider a lookup for the key 001100. The binary search algorithm rst tries the 3-bit pre x 001* (internal node), then the 5-bit pre x 00110* (not an internal node), and then nally the 4-bit pre x 0011*, which is the required leaf node. Pseudocode for this algorithm is given below.

[4]Assignment implies that the peer maintains the state as-sociated with the PHT node assigned to it. Henceforth, the discussion will use PHT node to also refer to the peer as-signed that node.

```
Algorithm: PHT-LOOKUP-BINARY
input  : A key K
output: leaf (K )

lo     0;
hi      D;
while (lo hi) do mid (lo +
    hi)/2;
    /*P_mid (K ) denotes prefix of K of length mid
        */
    node       DHT-LOOKUP(P_mid (K ));
    if (node is a leaf node) then return node ; else
        if (node is an internal node) then lo mid +
        1;
        else hi        mid- 1;
    end
end
return f ailure;
```

Binary search reduces the number of DHT lookups from (D + 1) to blog (D + 1)c + 1 log D. Never-theless, linear search is still signi cant for atleast two reasons. First, observe that the (D + 1) DHT lookups in linear search can be performed in **paral-lel**, as opposed to binary search, which is inherently sequential. This results in two modes of operation viz. low-overhead lookups using binary search, and low-latency lookups using parallel search. Second, binary search may fail ,i.e., be unable to correctly locate the leaf node, as a result of the failure of an internal PHT node [5] . On the other hand, lin-ear search is guaranteed to succeed as long as the leaf node is alive, and the DHT is able to route to it, and therefore provides a failover mechanism. Note that both algorithms are contingent on the fact that the DHT provides a mechanism to locate any PHT node via a single lookup.

2.2.2 Range Query

Given two keys L and H (L H), a range query returns all keys K contained in the PHT satisfying L K H . Range queries can be implemented in a PHT in several ways; we present two simple algorithms.

[5] Binary search will not be able to distinguish between the failure of an internal node and the absence of an internal node.

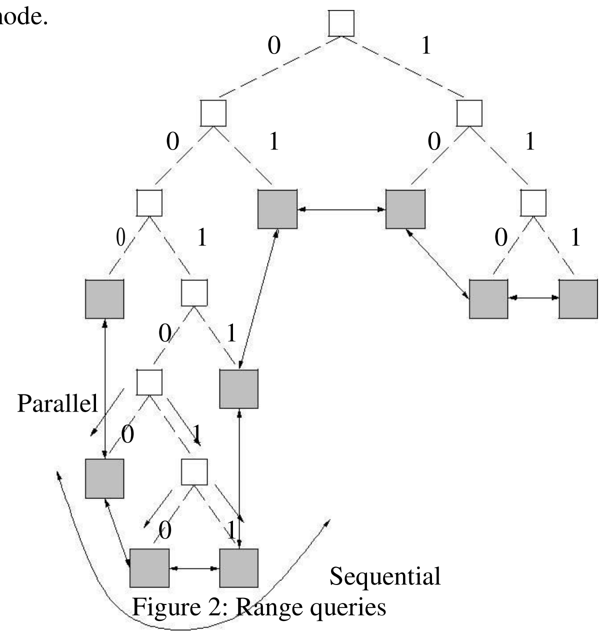


Figure 2: Range queries

The rst algorithm is to locate leaf (L) using the PHT lookup operation. Now the doubly linked list of threaded leaves is traversed sequentially until the node leaf (H) is reached. All values satisying the range query are retrieved. This algorithm is simple and e cient; it initially requires log D DHT lookups to locate leaf (L). It cannot avoid travers-ing the remaining nodes to answer the query. The disadvantage of this algorithm is that a sequential scan of the leaf nodes may result in a high latency before the query is completely resolved.

The second algorithm is to parallelize. Using the DHT, locate the node whose label corresponds to the smallest pre x range that completely covers the speci ed range. If this is an internal node, then re-cursively forward the query onward to those chil-dren which overlap with the speci ed range. This process continuues until the leaf nodes overlapping with the query are reached. If this is not an inter-nal node, the required range query is covered by a single leaf node, which can be located by binary search.

Figure 2 shows an example of range search. Con-sider a query for the range [001001; 001011]. In the sequential algorithm, a PHT lookup is used to lo-cate the node containing the lower endpoint, i.e., node 00100 . After this a traversal of the linked list forwards the query to the next two leaves 001010 and 001011 , which resolves the query. In the par-allel algorithm, we rst identify the smallest pre x range that completely covers the query, which is
0010 . A single DHT lookup is used to directly jump to this node, after which the query is for-warded in parallel within the sub-tree, until all leaf nodes that overlap with the search range are reached.

Note that in the parallel algorithm, it is sometimes desirable to break the search query into two, and treat these sub-queries independently. For exam-ple, a very small range that contains the midpoint of the space, will result in being the smallest pre x range containing it, thereby potentially over-loading the root. To prevent this, we observe that every range is contained in the union of two pre-x ranges that are of roughly the same size as the query (within a factor of 2). By handling these separately, it is possible to ensure a search starts at a level in the PHT that is appropriate for the query i.e. smaller queries start lower down in the PHT.

2.2.3 *Insert / Delete*

Insertion and deletion of a key K both require a PHT lookup operation to rst locate the leaf node leaf (K). Insertion of a new key can cause this leaf node to split into two children, followed by a redistribution of keys. In most cases, the (B + 1) keys are distributed among the two children such that each of them stores atmost B. However it is possible that all (B + 1) keys are distributed to the same child, necessitating a further split. In the worst case, an insertion can cause splits to cascade
all the way to a depth D [6], making insertion costs proportional to D. Similarly, in the worst case, deletion can cause an entire sub-tree of depth D to collapse into a single leaf node, incurring a cost proportional to D.

It is possible to reduce update costs and avoid problems of multi-node coordination through **stag-gered updates**. Only one split operation is allowed per insertion, and similarly, only one merge oper-ation is allowed per deletion. While this results in update costs reducing to log D DHT lookups (the cost of a PHT lookup to locate the leaf node), it also allows invariants 3 and 4 to be violated. A leaf node can now store upto (B + D) keys. This is not likely to be a problem because in most practical scenarios, B >> D.

2.3 Tries versus Trees

This section compares the merits of a trie-based in-dex, such as the PHT, with balanced tree-based in-dices, such as the B-tree, with particular emphasis on implementation in a distributed setting. This paper has described how the PHT data structure can be built over a DHT; it is likewise conceivable that a B-tree could be built over a DHT, with the

[6] This process must terminate because in the worst case, all keys are identical, and it is assumed that identical keys are distinguished by padding random bits at the end, and appropriately increasing D.

DHT being used to distribute B-tree nodes across peers in the network. While the tree-based indices may be better in traditional indexing applications like databases, we argue the reverse is true for im-plementation over a DHT.

The primary di erence between the two approaches is as follows: a trie partitions the **space** while a tree partitions the **data set**. In other words, a trie node represents a particular region of space, while a tree node represents a particular set of keys. Because a trie uses space, which is constant independent of the actual data set, there is some implicit knowl-edge about the location of a key. For example, in a trie, a key is always stored at a pre x of the key, which makes it possible to exploit the mechanism the DHT provides to locate a node via a single DHT lookup. In a tree, this knowledge is lacking, and it not possible to locate a key without a top-down traversal from the root. Therefore, a tree index **cannot** use the random access property of the DHT in the same manner. This translates into several key advantages in favor of the PHT when compared to a balanced tree index.

2.3.1 *Efficiency*

A balanced tree has a height of log N , and therefore a key lookup requires log N DHT lookups. In addi-tion, updates may require the tree to re-balanced. The binary search lookup algorithm in the case of the PHT requires only log D DHT operations, and updates have the same cost as well. Comparing the cost of lookups in the case of an index consisting of a million 32-bit keys, a tree index would require 20 DHT lookups as compared to 6 for the PHT to retrieve a key. Of course, multiway indexing could be used to reduce the height of the tree, but this would also leave the tree more vulnerable to faults in the indexing structure.

2.3.2 *Load Balancing*

As mentioned before, every lookup in a tree must goes through the root, creating a potential bottle-neck. In the case of a trie, binary search allows the load to be spread over $2^{\frac{D}{2}}$ nodes (assuming uniform lookups), thus eliminating any bottleneck.

2.3.3 Fault Resilience

In a typical tree-based structure, the loss of an in-ternal node results in the loss of the entire sub-tree rooted at the failed node. PHT however does not require top-down traversals; instead one can directly \jump" to any node in the PHT. Thus the failure of any given node in the PHT does not af-fect the availability of data stored at other nodes. In some sense, the indexing state in the trie is used only as an optimization. For example, observe that correct operation of the PHT is achievable using only the integrity of the doubly-linked list of leaf nodes[7]. Both updates (through linear search) and range queries (through sequential traversal of the list) can be handled without the help of the trie indexing structure. Contrast with a tree where the indexing structure is indispensible for both updates and queries, and is therefore vulnerable to failures.

2.4 PHT Enhancements

Until this point, we have discussed the use of PHTs for satisfying unidimensional range queries. In this section, we describe two re nements: functionality extensions to support multi-dimensional searching, and performance enhancements for scenarios with known or relatively static data distributions.

2.4.1 Multi-dimensional Indexing via Linearization

There are a plethora of centralized indexing schemes for supporting multidimensional range and near-neighbor queries; multiple surveys have been pub-lished in this area (e.g., [11, 28]). One class of heuristic multidimensional indexing schemes maps multidimensional data to a single dimension. This approach is sometimes called **linearization**, or **space-lling curves**, and well-known examples include the Hilbert, Gray code, and \Z-order" curves [16]. A multidimensional query is mapped to a unidimen-sional range query that spans from the lowest to highest linearization points of the original query. In general, linearized queries return a superset of the matching data, which has to be post- ltered. Recent linearization schemes like the Pyramid [3] and iDistance [35] techniques have been shown em-pirically to outperform traditional space- lling curves as well as popular multidimensional tree structures in high-dimensional scenarios.

Though work on multidimensional indexing via lin-earization schemes is largely heuristic, it has a strong practical attraction: linearization can be imple-mented as an overlay upon existing unidimensional range search structures, which are typically more frequently implemented and carefully debugged than specialized multidimensional indexes. This argu-ment holds for PHTs as well as for any distributed range search technique. PHTs have the added ad-vantage that their underlying substrate, DHTs, are rapidly emerging as a leading distributed building block.

After mapping a d-dimensional query Q_d into a unidimensional query Q^0, a PHT nds the answer set in O(log D + djQ^{0}j=Be) network hops, where jQ^{0}j is the size of the result set returned for the unidimensional query Q^0. Note that we have given no bound on the di erence between jQ^{0}j and jQ$_d$j,
and in the worst case jQ^{0}j = n. This di erence captures the ine cacy of the chosen linearization scheme; it is not particular to PHTs per se.

[7] This is somewhat similar to Chord whose correct operation depends only on the integrity of the successor pointers

2.4.2 Indexing Known Distributions

Relative to tree-based indexes, a disadvantage of PHTs is that their complexity is expressed in terms of the log of the domain size, D, rather than the size of the data set, N . In many scenarios, however, data is from a known distribution: for example, keywords in text search follow well-known Zip an distributions, and range search queries (e.g. text-
pre x queries like \Cali*") are quite natural. Here we informally argue that for known distributions, the PHT can be modi ed to run in O(log log N) expected hops.

We begin by examining the simple uniform distri-bution over D-bit keys. For N data items drawn from this distribution, the expected depth of a leaf in the PHT is O(log N), with low variance. Hence for uniform distributions, the expected number of hops with PHT-LOOKUP-BINARY is O(log log N). This search algorithm can be improved further via a search algorithm that starts at pre x-length log N , and proceeds upward or downward as necessary.

For known but non-uniform distributions, similar performance can be achieved by \warping" the space appropriately, remapping the original distribution to a uniform distribution. For each data point drawn from the original distribution, its rst bit is remapped to be 0 if it is lower than the mid-point of the distribution's PDF, and 1 otherwise; this assignment proceeds recursively through all D bits. The resulting set of mapped points are essentially drawn from the uniform distribution, and a PHT built on these points will have path lengths as described above. Queries in the original space are mapped accordingly, and will perform with O(log log N) expected hops.

For globally well-known distributions (e.g. terms in spoken English), the warping function is more or less xed and can be distributed as part of the PHT code. However, in practice many data sets come from distributions that do change, but quite slowly. For example, the distribution of terms in lesharing applications will shift slightly as pop-ularity shifts; some terms may suddenly become popular when a new popular le is released, but most terms' frequency will remain relatively static for long periods of time. For such slowly-changing distributions, a gossip-based scheme can be used to disseminate compressed representations of the distribution to all nodes, and any changes to the distribution can be re ected via periodic remap-ping of the data and queries (perhaps as part of soft-state refresh.)

We close this section by observing a general dual-ity. A tree index is actually quite analogous to the pairing of a trie-based scheme with a known dis-tribution: the \split-keys" in a tree index capture the data distribution, and the pointers in the tree index serve the same function as the bits in the trie encoding. An advantage of the PHT approach is the ability we noted above to \jump" into any point of the trie via hashing; global knowledge of the dis-tribution provides this uniformity in addressing. A similar trick could be achieved in tree indexes as well, if every searcher had a fair approximation of the split keys in the tree and the locations of the tree nodes in an identi er space.

3. EVALUATION

This section presents a simulation-based evaluation of the PHT data structure. Although these simu-lation results are by no means comprehensive, we present them as preliminary experimental evidence that the PHT is a viable solution. A more complete evaluation, along with gaining experience with de-ployment of a working prototype, is the focus of our current e orts.

Our simulation setup is as follows. A PHT that indexes 30-bit keys is created on top of a DHT consisting of 1000 nodes. Our focus is on evalu-ating the performance of the data structure; for that reason we abstract away many of the details of the DHT by using a stripped-down version of the Chord protocol. 2^{16} arti cially generated keys are inserted into the PHT coming from a uniform distribution over the entire 2^{30} keyspace. We use an arti cially low block size of B = 20 in order to generate a non-trivial instance of the PHT.

3.1 Range queries

Recall that a range query is evaluated by travers-ing the leaf nodes of the PHT. The complexity (and latency) of the range query operation depends on the number of such leaves, which is a function of the output, i.e., how many keys actually satisfy the given query. In the ideal case, if the output size is O and the block size is B, the number of nodes traversed should be about d $^{O}_{B}$ e. To see how well the PHT distributes keys among the leaf nodes, we generate 1000 randomly generated queries of size varying from 2^{22} to 2^{26} , measured how many leaf nodes were required to be traversed. The results normalized to the optimal number d $^{O}_{B}$ e are shown in figure 3. The number of leaf nodes required to be traversed is roughly the same in all cases: about 1.4 times the optimal value. To evaluate the e ect of skewed distributions on the PHT structure, this experiment was repeated with a Gaussian distri-bution centered at the midpoint of the space to generate input keys. For ranges that are close to the mean, where keys are densely clustered, the PHT does well, actually out-performing the uni-form case. For sparser regions, the PHT does not do as well, but no worse than 1.6 the optimal value. These results indicate that the PHT incurs only a reasonably small constant factor of overhead (in terms of nodes visited) more than the theoretically optimal value.

3.2 Load balance

The next experiment attempts to verify the asser-tion that the PHT spreads network load evenly, and therefore does not have a bottleneck, unlike a binary tree. To test this hypothesis, we generated 100,000 PHT lookups on uniformly distributed keys and observed the distribution of lookup tra c. By lookup tra c, we mean the DHT queries generated by the binary search algorithm, and not the under-lying DHT routing tra c. Figure 4 shows the dis-tribution of lookup tra c over all the nodes in the DHT. It can be seen that about 80 % of the nodes see less than 400 lookups (out of 100,000). The rest of the nodes, which correspond to PHT leaf nodes, receive more tra c, but in no case higher than 1800. Contrast this with a B-tree where each of the 100,000 messages must necessarily go through the root. To test the e ect of network size, the experiment was repeated for 1000, 2000 and 3000 nodes respectively. As expected, a larger number of nodes reduces the amount of per-node tra c, as PHT pre xes are distributed among more nodes. However the actual PHT leaf nodes continue to re-ceive higher amounts of tra c than the rest of the nodes.

4. RELATED WORK

Building e cient data structures for searching is one of the fundamental problems in computer sci-ence; [19, 8] are good references. Our PHT pro-posal is particularly reminiscent of Litwin's Trie Hashing [21], but has an added advantage that the \memory addresses" where buckets of the trie are stored are in fact the DHT keys obtained by hash-ing the corresponding pre xes.

With respect to the problem of implementing range search over peer-to-peer systems, Aspnes and Shah [1] have proposed **skip graphs**, a distributed data structure based on the skiplist that provides a range search solution. However they do not provide a mapping from keys to peers in a network; such a mapping is provided by Awerbuch and Scheideler

In recent work, Karger and Ruhl [17] propose a randomized protocol called **item balancing** that bal-ances the distribution of items by having DHT nodes adaptively change their identi ers. While providing excellent theoretical properties, their so-lution relies on more than just the hashtable in-terface of the underlying DHT, which could poten-tially create a barrier to deployment. A related protocol has been proposed by Ganesan and Bawa [12].

Other related work includes a DHT-based caching scheme [13] and a technique speci cally for the CAN DHT based on space- lling curves [32].

Cone [4] is a trie-based data structure that is used to evaluate aggregation operators, such as MIN, MAX and SUM, over keys in a DHT. Although the PHT is also based on a trie, it di ers from Cone in three signi cant respects. First, Cone builds a trie over uniformly distributed node identi ers. Second, Cone does not support range queries. Fi-nally, Cone is a DHT augmentation where as the PHT builds on top of the DHT.

Waldvogel et al [33] have proposed an IP lookup algorithm based on binary search of pre xes orga-nized into hashtables based on pre x length. Al-though they are solving **longest pre x match**, a di erent but related problem, their binary search technique is similar to the PHT lookup algorithm. The key distinguishing characteristic is that the PHT operates in a distributed setting, with an entirely di erent set of constraints and issues, as opposed to an IP lookup algorithm that is imple-mented in hardware in a high-speed router.**5.**

CONCLUSION

In their short existence, DHTs have become a widely used tool for building large-scale distributed sys-tems. While the lookup interface o ered by DHTs is broadly applicable, it does not naturally support a very common feature in database and other in-formation processing systems: range queries. Our goal was to address this shortcoming but, contrary to early e orts in the eld, subject to the constraint that these queries only use the lookup interface and not rely on changes to or knowledge of the under-lying DHT routing algorithm. This would ensure that the solution would apply to any DHT, not just those speci cally engineered for the task. To this end, we presented the design and evaluation of Pre-x Hash Trees (PHT), a data structure designed to support range queries. PHT has the properties tra-ditionally required of large-scale Internet systems: self-organizing, scalable, and robust in the presence of failures. While it does not prevent loss of data due to node outages, such failures do not prevent it from producing results from the other nodes.

In short, we believe that PHT will enable general-purpose DHTs to support a wider class of queries, and then broaden the horizon of their applicability.

6. REFERENCES

[1] Aspnes, J., and Shah, G. Skip graphs. In **Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms** (Baltimore, MD, Jan 2003).

[2] Awerbuch, B., and Scheideler, C.

Peer-to-peer Systems for Pre x Search. In **ACM Symposium on Principles of Distributed Computing** (Boston, MA, July 2003).

[3] Berchtold, S., Bohm,• C., and Kriegel, H.-P. The pyramid-technique: Towards breaking the curse of dimensionality. In **Proc.**

ACM SIGMOD International Conference on Management of Data (Seattle, WA, June 1998), pp. 142{153.

[4] Bhagwan, R., Voelker, G., and Varghese, G. Cone: Augmenting DHTs to Support Distributed Resource Discovery. Tech. Rep. UCSD CS2002-0726, Computer Science Department, University of California, San Diego, November 2002.

[5] Blake, C., and Rodrigues, R. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In **HotOS IX** (May 2003).

[6] Cabrera, L., Jones, M. B., and Theimer, M. Herald: Achieving a Global Event Noti cation Service. In **HotOS VIII** (May 2001).

[7] Clip2. The Gnutella Protocol Speci cation v.0.4, March 2001.

[8] Cormen, T., Stein, C., Rivest, R., and Leiserson, C. **Introduction to Algorithms**. McGraw-Hill Higher Education.

[9] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area Cooperative Storage with CFS. In

Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001) (Lake Louise, AB, Canada, October 2001).

[10] Freedman, M. J., Freudenthal, E., and Mazieres, D. Democratizing Content Publishing with Coral. In **Proceedings of the**

USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2004 (San Francisco, CA, March 2003).

[11] Gaede, V., and Gunther,• O.

Multidimensional access methods. **ACM Computing Surveys 30**, 2 (June 1998), 170{231.

[12] Ganesan, P., and Bawa, M. Distributed Balanced Tables: Not Making a Hash of it All. Tech. rep., Computer Science Department, Stanford University, 2003.

[13] Gupta, A., Agrawal, D., and Abbadi, A. E. Approximate Range Selection Queries in Peer-to-Peer Systems. In **Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)** (Asilomar, CA, January 2003).

[14] Hildrum, K., Kubiatowicz, J. D., Rao, S., and Zhao, B. Y. Distributed Object Location in a Dynamic Network. In **14th**

ACM Symposium on Parallel Algorithms and Architectures (Aug. 2002).

[15] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. Querying the Internet with PIER. In **Proceedings of VLDB 2003** (Berlin, Germany, September 2003).

[16] Jagadish, H. V. Linear clustering of objects with multiple atributes. In **Proc. ACM SIGMOD International Conference on Management of Data** (Atlantic City, NJ, May 1990), pp. 332{342.

[17] Karger, D., and Ruhl, M. Simple E cient Load Balancing Algorithms for Peer-to-Peer Systems. In **Proceedings of the Third International Peer-to-Peer Systems Workshop (IPTPS)** (Feb. 2004).

[18] Karp, B., Ratnasamy, S., Rhea, S., and Shenker, S. Spurring Adoption of DHTs with OpenHash, a Public DHT service. In

Proceedings of the Third International Peer-to-Peer Systems Workshop IPTPS 2004
(Feb. 2004). Knuth, D. **The Art of Computer Programming**. Addison-Wesley.

[19] Li, J., Loo, B. T., Hellerstein, J. M., Kaashoek, F., Karger, D., and Morris,

R. On the Feasibility of Peer-to-Peer Web Indexing and Search. In **Proceedings of the**

2nd International Workshop on Peer-to-Peer Systems (IPTPS). (Berkeley, CA, Feb. 2003).

[20] Litwin, W. Trie Hashing. In **Proceedings of ACM SIGMOD** (Ann Arbor, MI, 1981).

[21] Malkhi, D., Naor, M., and Ratajczak,

D. Viceroy: A Scalable and Dynamic Emulation of the Butter y. In **ACM**

Symposium on Principles of Distributed Computing (July 2002).

[23] Maymounkov, P., and Mazieres, D.

Kademlia: A peer-to-peer information system based on the xor metric. In **1st International**

Workshop on Peer-to-Peer Systems
(Cambridge, MA, Mar. 2002).