LHT: A Low-Maintenance Indexing Scheme over DHTs

Yuzhe Tang and Shuigeng Zhou
Department of Computer Science and Engineering and
Shanghai Key Lab of Intelligent Information Processing
Fudan University, Shanghai 200433, China
{yztang, sgzhou}@fudan.edu.cn

Abstract

DHT is a widely-used building block in P2P systems, and complex queries are gaining popularity in P2P applications. To support efficient query processing over DHTs, effective indexing structures are essential. Recently, a number of indexing schemes have been proposed. However, these schemes have focused on improving query efficiency, and as a trade-off, sacrificed maintenance efficiency — an important performance measure in the P2P context, where frequent data updating and high peer dynamism are typically incurred. In this paper, we propose LHT, a Low maintenance Hash Tree, for efficient data indexing over DHTs. LHT employs a novel naming function and a tree summarization strategy to gracefully distribute its index structure. It is adaptable to any DHT substrates, and is easy to be implemented and deployed. Experiments show that in comparison with the state-of-the-art indexing technique, LHT saves up to 75% (at least 50%) maintenance cost, and achieves better performance for exact-match queries and range queries.

1 Introduction

Distributed Hash Table (DHT) is a widely-used building block in Peer-to-Peer (P2P) systems. DHTs employ consistent hashes [14] to map both data and peers to an *identifier space*, on which the overlay is established. Based on different overlays, various DHT substrates have been proposed, such as Chord [22], CAN [17], Pastry [20] and Tapestry [23]. DHTs have several outstanding advantages: 1) Scalability and efficiency. In a typical DHT of N peers, the lookup latency is $O(\log N)$ hops with each peer maintaining only $O(\log N)$ "neighbors"; 2) Robustness. DHTs are resistant to node failures that are common in large-scale P2P networks; 3) Load balance. Due to uniform hashes, storage load balance in DHTs can be easily achieved.

But we may have too much of a good thing — basic DHTs do not support complex query processing. Because complex queries typically require preservation of data locality, which however is destroyed by the uniform hashes in DHTs. As a result, semantically proximate data may not be stored closely, and various complex queries, like range queries can not be efficiently supported. However, complex queries are highly desired and actually gaining popularity in many P2P applications. For example, users of a P2P file sharing system may want to "find all MP3 files published between Jan. 1, 2007 and now", which actually demands a range query. So, the popularity of complex queries poses an urgent need for DHT-based indexing schemes.

Generally speaking, to design an indexing structure, query efficiency comes as the first priority. But in P2P networks, maintenance efficiency also turns out to be a critical factor. Because peers frequently join/leave the networks, which incurs a large amount of data insertion/deletion. As a result, P2P systems have to invest a lot maintenance cost for adjusting their index structures. This problem, however, has not be effectively resolved in the existing P2P indexing schemes. Instead, they focused on improving query efficiency, and as a trade-off, sacrificed maintenance efficiency. More specifically, in a P2P network, each peer maintains a local view of global index structure. To achieve better query performance, a possible way is to augment the local view and let each peer know more about the global indexing tree. For example, in Prefix Hash Trie (PHT) [16, 4], each leaf knows its neighboring leaves. In Distributed Segment Tree (DST) [24], each tree node knows how many items are indexed by its left/right child. The Range Search Tree (RST) [9] goes to extreme, which gives each tree node the entire knowledge of global index tree. When updating the global tree, a node splitting can cause a broadcasting to all tree nodes, incurring extremely high bandwidth cost. Another reason for maintenance inefficiency of existing indexing schemes is that their index structures are not gracefully distributed over DHTs. As a result, although better query performance is achieved, the holistic performance (with respect to both query processing and indices maintenance)

1063-6927/08 $25.00 © 2008 IEEE
DOI 10.1109/ICDCS.2008.61

gets no improvement.

Based on this observation, in this paper we propose LHT, a *L*ow maintenance *H*ash *T*ree for data indexing in DHT based P2P systems. LHT requires no modification of the underlying DHTs and can be easily adapted to any DHT substrate. There are two novelties in LHT: a naming function that gracefully distributes the index structure over the underlying DHT, and a local tree summarization strategy that offers each peer a local view of the index tree, and requires no extra maintenance. LHT can efficiently support different complex queries, including range queries and min/max queries etc.

In summary, contributions of this paper are as follows.

- LHT — A low maintenance indexing scheme over DHTs is proposed. To best of our knowledge, this is the first work explicitly addressing the need of low maintenance cost in over-DHT indexing schemes.
- Algorithms for exact-match queries, range queries and min/max queries on LHT are developed.
- A cost model for quantifying indexing maintenance cost is given, which is applicable to LHT and other over-DHT indexing schemes.
- Extensive experiments are conducted to evaluate LHT performance. In comparison with PHT, a state-of-the-art indexing scheme with respect to maintenance efficiency, LHT can save up to 75% (and at least 50%) overhead, and achieve better query performance.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 presents the LHT index structure. Section 4 describes how LHT distributedly grows with data insertion. Section 5, 6, 7 respectively present the algorithms of LHT operations of lookup, range query and min/max query. Section 8 analyzes LHT maintenance cost. Section 9 describes experimental results. Section 10 concludes this paper.

2 Related Work

In recent years, a number of P2P indexing schemes have been proposed, which largely fall into two categories: DHT based and non-DHT based.

DHT-based indexing schemes, termed *over-DHT paradigm*, are built as an incremental module over DHTs. These schemes rely only on the "put/get" interface of generic DHTs, and can be easily adapted to various DHT substrates. As a typical DHT-based indexing structure, PHT [16, 4] uses a trie structure to index one dimensional discrete data. For efficient query processing, PHT maintains B+ tree links between neighboring leaves. Distributed Segment Tree (DST) [24] supports efficient range queries and cover queries. It replicates data keys across all ancestors of a leaf, and leverages parallel lookups to reduce query latency. Due to replication, data insertion in DST is inefficient. RST [9] follows a similar philosophy: it replicates data to all ancestors, and the tree structural information to all tree nodes. With index tree globally known, RST achieves one-hop exact-match query and efficient range query, but at the expense of high maintenance cost. A single leaf splitting could lead to a broadcasting to all nodes, which is quite inefficient and unscalable in P2P networks. With a similar tree maintenance to RST, DKDT [10] embeds the k-d tree to support similarity search over DHTs. PRISM [21] employs reference vectors to generate DHT keys for multi-dimensional objects and supports similarity search over DHTs. Chen et al. [5] presented a framework for range indexing and discussed various strategies for mapping tree-based index structures onto DHTs.

An alternative of data indexing with DHTs is to replace the uniform hash with LSH, the Locality Sensitive Hash. Thanks to LSH's locality preservation, some DHTs can directly index data on overlays and support efficient range query processing [8, 15]. Unlike over-DHT schemes, these schemes usually rely on a specific overlay and can not enjoy a wide deployment. Gupta et al. [11] applied LSH to mapping ranges to a DHT, which provides approximate answers to range queries. LSH-Forest [2] eliminates LSH's data-dependent parameters and is applied to P2P systems. In contrast to traditional DHTs, DHTs with LSH have to sacrifice their load balance.

The non-DHT based indexing schemes, termed *substrate-dependent paradigm*, make no use of DHTs and design their own substrates based on various data structures. Skip graph [1] is a distributed range queryable structure based on skip lists. BATON [12] is an overlay organized as a balanced binary tree. VBI-Tree [13] is a framework that aims at mapping any existing index tree onto BATON. It indexes multi-dimensional data and supports range queries and KNN queries. PTree [6] and PRing [7] are distributed B-trees on P2P networks. Mercury [3] uses a hierarchical ring structure to index multi-dimensional data. The substrate-dependant schemes provide efficient query processing, but rely on specific substrates, which weakens their adaptability. And designing such indexing schemes should deal with some low level issues, which complicates the design and implementation process. For example, they need a non-trivial extension for peer load balance.

3 The LHT Index Structure

In this section, we describe the LHT index structure and the method of mapping LHT to the underlying DHT.

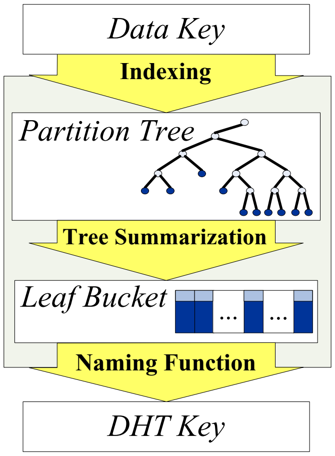


Figure 1: LHT Indexing Architecture

3.1 Data Model and System Architecture

In LHT, a data unit is called a *record* and each record is identified by a distinct value, called *data key* δ. For example, in a P2P database, a tuple can be seen as a record, and any candidate key could be its data key. Basic LHT deals with one dimensional data[1], i.e., the data key δ is a numerical value in an interval. Without loss of generality, we assume that δ is a real value falling in $[0, 1]$.

To assign records to the underlying DHT, each data record has an additional key, named *DHT key* κ. Given a DHT key κ, the corresponding record is mapped to a DHT peer whose identifier is closest to but smaller than $hash(\kappa)$, i.e., the one responsible for $hash(\kappa)$. In the raw DHT (where no indexing scheme is deployed), κ simply equals δ; while in LHT, for preserving data locality, there exists a *data-to-DHT transform* from δ to κ. In Fig. 1, the LHT indexing architecture illustrates how the data-to-DHT transform works. First, LHT employs a *space partition tree* to index data. Then, with this tree distributed and summarized in a data structure called *leaf bucket*, LHT leverages a novel naming function to map leaf buckets to the underlying DHT.

3.2 Space Partition Tree

We start from a centralized viewpoint: LHT has an index structure, called space partition tree (or *partition tree* for short). Fig. 2 illustrates how the space partition tree indexes data. The bottom histogram represents data distribution, and the shape of partition is well adapted to it. Essentially, the space partition tree is a binary tree with following structural properties:

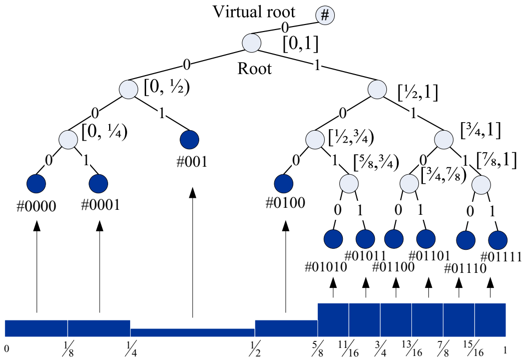
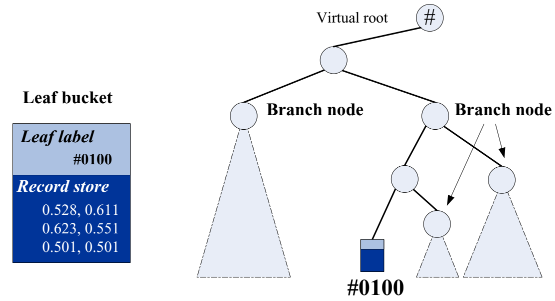


Figure 2: An example of space partition tree

- (Double-root) Space partition tree has two roots. The additional root, termed *virtual root*, is a virtual node above the regular one.
- (Fullness) Each tree node, except the virtual root, has 0 or 2 children, i.e., all internal nodes have 2 children. This property and the double-root feature together guarantee that the number of leaf nodes equals the number of internal nodes.
- (Record storage) Data records are stored only in leaf nodes, and each leaf can store at most θ_{split} records. θ_{split} is a pre-set threshold for leaf splitting: when the number of records in one leaf exceeds θ_{split}, the leaf splits; and whenever an internal node's subtree contains less than θ_{split} records, all leaves in this subtree are merged into one node.
- (Space partition strategy) To index data, the tree partitions data space continuously. Specifically, each leaf indexes an interval. When splitting a leaf, the partition point is the interval's median, which is unrelated to data distribution. For example, in Fig. 2 the partition point of the root is always $\frac{1}{2}$ (the median of $[0, 1]$), even when data keys are not evenly distributed between $[0, \frac{1}{2})$ and $[\frac{1}{2}, 1)$. This space partition strategy makes each tree node's interval globally known, which is essential in distributed system.

Each node in the tree is given a unique label. The virtual root is labeled with a special character, say "#" in this paper. Each edge is labeled with a binary number, 0 for the edge connecting to the left child, and 1 otherwise. As a special case, the edge between the virtual root and the regular root is labeled with 0. And for any tree node, its label is the concatenation of binary numbers in the path from the node to the virtual root. We proceed to define the notation for describing the partition tree: λ denotes the label of a leaf node, and ω denotes the label of an internal node; Λ denotes

[1] One dimensional index structure can serve as an infrastructure for multi dimensional indexing (e.g., by using SFC [4]).



(a) Data structure (b) Local tree of #0100

Figure 3: Leaf bucket and local tree

the set of all leaves' labels, i.e., $\Lambda = \{\lambda\}$; and similarly Ω denotes the set of all internal nodes' labels, i.e., $\Omega = \{\omega\}$.

3.3 Local Tree Summarization

To materialize the partition tree, we need only to map leaf nodes to the underlying DHT. Note that all internal nodes are empty and only leaf nodes store data; and the raw leaf nodes lack the knowledge of the tree structure, which as we will see, is critical to distributed query processing. We propose a distributed data structure, termed *leaf bucket*, to store data records and summarize the partition tree's structural information.

Each leaf bucket corresponds to a distinct leaf node. As illustrated in Fig. 3a, one bucket consists of two fields: *leaf label* that maintains its label λ, and *record store* that contains all related data records. For each bucket, the label λ provides a local view of the partition tree, which is called *local tree*. As shown in Fig. 3b, the local tree of leaf #0100 consists of all of its ancestors and the related *branch nodes* around these ancestors. The label of any node in the local tree can be inferred directly from λ, because each ancestor's label is a prefix of λ; and each branch node has its label consisting of two parts: a prefix of λ and a "0" (or "1") as its final bit. Due to the tree's fullness, every branch node must exist, and holds a subtree called *neighboring tree*, as represented by the dotted triangles in Fig. 3b. The depth of a neighboring tree is unknown from the current local tree, but it may be inferred from some other leaf's local tree.

In a global viewpoint, the union of all local trees guarantees the partition tree's integrity. In other words, the leaf labels (buckets) together summarize the tree's structural information. Since leaf buckets cover the knowledge of partition tree, we map each bucket as an atomic unit into the underlying DHT.

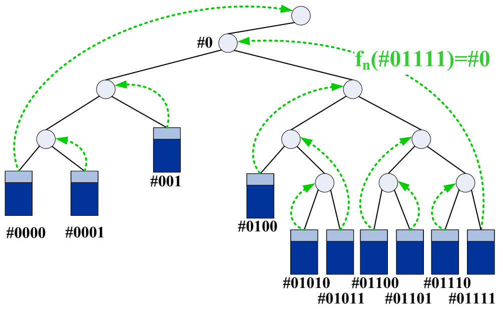


Figure 4: Naming function in LHT

3.4 Naming Function

For a bucket with label λ, the naming function f_n generates its DHT key, i.e., $\kappa = f_n(\lambda)$. f_n is defined as below.

Definition 1. *For any leaf label $\lambda \in \Lambda$, the **naming function** is*

$$f_n(\lambda) = \begin{cases} p0 & \textit{if } \lambda = p011*, \\ p1 & \textit{if } \lambda = p100*, \\ \# & \textit{if } \lambda = \#00*. \end{cases}$$

where $p = \#0[0|1]*$ or $\#$[2]*. That is, if λ ends up with 0, the naming function f_n truncates all consecutive 0s from λ's end. Otherwise, it truncates the consecutive 1s. For example, $f_n(\#01100) = \#011$, $f_n(\#01011) = \#010$.*

From a tree's perspective, each λ represents a leaf node, and interestingly, each $f_n(\lambda)$ represents a distinct internal node. Fig. 4 depicts such a mechanism, in which each leaf bucket λ is "named" to an internal node $f_n(\lambda)$ by a dotted arrow, like $f_n(\#01111) = \#0$. This nice property is originated from the double-root and fullness of the partition tree. Recall that Λ and Ω represent the sets of labels of internal nodes and leaves, respectively, we have the following theorem.

Theorem 1. *f_n is a bijective mapping from Λ to Ω.*

Proof. We first prove that f_n is indeed a mapping from Λ to Ω, and then prove that f_n is bijective.

For $\forall\lambda \in \Lambda$, $f_n(\lambda)$ is a prefix of λ. By the labeling strategy, λ's any prefix represents an ancestor of the corresponding leaf, and thus represents an internal node. Therefore, f_n is a mapping from Λ to Ω.

As for the bijection, we prove a more concrete proposition that "for $\forall\omega \in \Omega$, there is one and only one λ mapped to it". For $\forall\omega \in \Omega$, there are two cases: ω ends up with 0 (i.e., $\omega = \omega'0$), or with $\hat{0}$ (i.e. $\omega = \omega'1$ or $\omega = \#$). For the

[2] Here we use the regular expression: $[0|1]$ means 0 or 1, and $*$ means repeating any time (including 0 time).

first case, the leaf that is mapped to ω must have a label as $\omega 11*$, because $f_n(\omega 11*) = f_n(\omega' 011*) = \omega' 0 = \omega$. By the labeling strategy, for a specific ω, there is one and only one leaf in Λ labeled as $\omega 11*$, i.e., the rightmost leaf in the subtree rooted at ω. Therefore, the proposition holds true for the first case. For the second case, it is the leaf as $\omega 00*$ that is mapped to ω, because $f_n(\omega 00*) = f_n(\omega' \hat{0} 11*) = \omega' \hat{0} = \omega$. Here, the special case of the virtual root, i.e., $\omega = \#$ is considered. Since there is one and only one leaf in Λ labeled as $\omega 00*$ (i.e., the leftmost leaf in the subtree rooted at ω), the proposition holds true for the second case. Altogether, f_n is a bijective mapping from Λ to Ω. □

Note that $f_n(\lambda)$ serves as the DHT key, Theorem 1 implies that the naming function actually organizes the internal structure of partition tree in the DHT key space.

4 Incremental Tree Growth

In the previous section, we describe the static mapping of LHT over DHTs. In this section, we introduce how LHT dynamically grows with data insertion. After data is inserted into LHT (the process of data insertion will be elaborated in Section 5), some leaf buckets may get saturated and then split. Each splitting produces two new leaf buckets. Among them, one stays on the current peer, denoted as *local leaf*, while the other, denoted as *remote leaf* is pushed out to another peer. We call this process *incremental tree growth*. Some part of a splitted leaf remains unchanged during tree growth. This is due to a nice property of the naming function, described in the following theorem.

Theorem 2. *If a leaf λ is splitted into two nodes, $\lambda 0$ and $\lambda 1$, the naming function will maps one node to $f_n(\lambda)$, and the other to λ.*

Proof. First consider the case that λ ends with 1. That is, $\lambda = p011*$. The naming function maps it to $f_n(\lambda) = f_n(p011*) = p0$. Now, the leaf λ splits into two nodes, $\lambda 0$ and $\lambda 1$. And the naming function maps them to,

$$\begin{cases} f_n(\lambda 0) = f_n(p011*0) = p011* = \lambda \\ f_n(\lambda 1) = f_n(p011*1) = p0 = f_n(\lambda) \end{cases}$$

For the second case, λ ends up with 0. We have $\lambda = p100*$ or $\lambda = \#00*$, for short, $\lambda = p\hat{0}00*$. The naming function maps it to $f_n(\lambda) = f_n(p\hat{0}00*) = p\hat{0}$. After λ's split, the naming function maps new leaf buckets to,

$$\begin{cases} f_n(\lambda 0) = f_n(p\hat{0}00*0) = p\hat{0} = f_n(\lambda) \\ f_n(\lambda 1) = f_n(p\hat{0}00*1) = p\hat{0}0* = \lambda \end{cases}$$

□

Theorem 2 directly leads to incremental tree growth. By LHT's mapping strategy, the original leaf bucket λ is assigned to the DHT peer with regard to $hash(f_n(\lambda))$. After splitting, one leaf bucket is still named to $f_n(\lambda)$, and thus remains on the same peer. This bucket is the local leaf. The other leaf bucket that is named to λ, is the remote leaf.

Algorithm 1 Leaf split(leaf bucket b)

```
1: $\lambda \leftarrow b.leaflabel$
2: if $\lambda = p011*$ then
3:     $rb.leaflabel \leftarrow \lambda 0$
4:     /*rb is the remote leaf bucket*/
5:     $b.leaflabel \leftarrow \lambda 1$
6: else
7:     $rb.leaflabel \leftarrow \lambda 1$
8:     $b.leaflabel \leftarrow \lambda 0$
9: Assign corresponding records to $rb$ and delete them in $b$.
10: Write $b$ back to the local disk.
11: DHT-put($\lambda$, $rb$)
```

Algorithm 1 describes how the leaf bucket λ splits in a distributed fashion. The function $leafsplit(b)$ is triggered whenever a bucket b contains more than θ_{split} records. It checks the value of current leaf label λ, and accordingly updates the labels, for b and the remote leaf bucket rb (lines 1–8). It then reassigns records between b and rb (line 9). After updating b in the local disk, the algorithm conducts a DHT-put to put rb to other peer (line 11). During the whole process, the algorithm uses only local knowledge and consumes one DHT-lookup (in DHT-put). Updating the leaf label and thus the local tree, requires no extra DHT-lookup.

5 LHT Lookup

For a data key δ, LHT lookup[3] returns the leaf bucket with label $\lambda(\delta)$, i.e., the bucket which covers δ. As a real number, the key δ can be converted into a binary string, long enough that any possible $\lambda(\delta)$ must be a prefix of it. For example, if we know in advance[4], that the possible maximum length of λ is 6, $\lambda(0.4)$ must be a prefix of the binary string $\#00110$ (with length 6). Here, $\#0$ is the root prefix, and 0110 is the binary number of 0.4. In Fig. 2, $\lambda(0.4) = \#001$. Note that λ's maximum length equals the LHT maximum depth plus 1. We denote this depth as D, the binary string as $\mu(\delta, D)$, and the set of all $\mu(\delta, D)$'s prefixes as $\Gamma(\delta, D)$ or shortly $\Gamma(\mu)$. So we have,

$$\lambda(\delta) \in \Gamma(\delta, D)$$

Note that $\Gamma(\delta, D)$ consists of prefixes of lengths from 2 to $D + 1$.

[3] In this paper, we may refer to LHT lookup as "lookup" for short, and for clarity the DHT-lookup always remains its full name.

[4] As in PHT, this priori knowledge can be obtained by estimating the size and the distribution of the data set indexed.

In order to find $\lambda(\delta)$ in $\Gamma(\delta, D)$, LHT conducts a binary search, as illustrated in Algorithm 2. It initiates an interval for candidate prefix length between 2 and $D+1$ (line 2). In each loop, it tries the median of the interval(line 4), and conducts a DHT-get for the corresponding name (line 6). If the DHT-get is failed, meaning that current prefix x is too long, it shortens the longer bound (line 9). Note that all prefixes between $f_n(x)$ and x are named to $f_n(x)$. There is no need to try any of these prefixes again, and the longer bound is set to $f_n(x)$. If the returned bucket covers δ, the algorithm returns the bucket name $f_n(x)$ (line 12). Otherwise (line 15), x itself represents an internal node in LHT and the shorter bound is reset to $f_{nn}(x, \mu)$, as defined blow.

Definition 2. *For $\forall x \in \Gamma(\mu)$, the* ***nextnaming function*** *$f_{nn}(x, \mu)$ is,*

$$f_{nn}(x,\mu) = \begin{cases} q1 & \textit{if } x = p0*0,\ q = x0* \textit{ and } \mu = q1[0|1]*, \\ q0 & \textit{if } x = p1*1,\ q = x1* \textit{ and } \mu = q0[0|1]*. \end{cases}$$

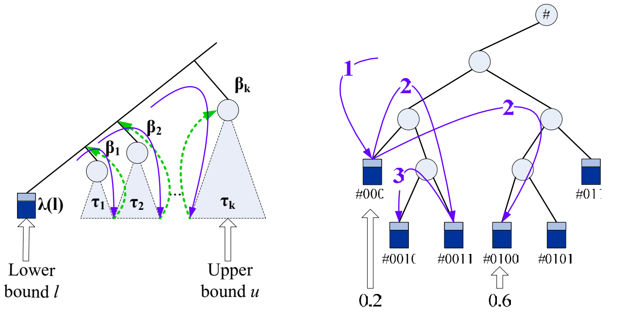
Note that x must be a prefix of μ. The next naming function finds the first bit that lies right of x and is not x's ending bit. For example, $f_{nn}(\#0011, \#0011100) = \#001110$.

The prefixes between x and $f_{nn}(x, \mu)$ share the same name, $f_n(x)$. For example, $f_{nn}(\#0011, \#0011100) = \#001110$, and $f_n(\#00111) = \#00 = f_n(\#0011)$. They must be named to the same DHT key, and thus there is no need to try them twice.

Algorithm 2 LHT-lookup(data key δ)

```
1:  $\mu$ ←binary-convert($\delta$)
2:  shorter← 2, longer←$D+1$
3:  while shorter ≤ longer do
4:      mid←(shorter+longer)/2
5:      x← $\mu$.prefix(mid)
6:      bucket← DHT-get($f_n$(x))
7:      if bucket=NULL then
8:          /*a failed DHT-get*/
9:          longer←$f_n$(x).length
10:     else if bucket contains $\delta$ then
11:         /*the target leaf bucket*/
12:         return $f_n$(x)
13:     else
14:         /*an internal node*/
15:         shorter← $f_{nn}$(x,$\mu$).length
16: return NULL
```

An example Consider a lookup of 0.9 with $D = 14$. Suppose LHT is as in Fig. 2 and the target bucket is leaf #01110. Note that $\mu(0.9, 14) = \#01110011001100$. LHT first tries the the prefix of middle length, i.e., #0111001, and conducts a DHT-lookup for $f_n(\#0111001) = \#011100$. It returns failure and next try is $f_n(\#011) = \#0$. The returned bucket is #01111 which does not contain 0.9. In this case, the shorter bound is reset to the length of $f_{nn}(\#011, \#01110011001100) = \#01110$. (note that the #0111 is also named to #0 and need not try again.) The next try is #01110, which reaches the target.



(a) Local tree of λ and the recursive forwarding

(b) An example

Figure 5: Range query processing

Complexity We measure the number of DHT-lookups as the complexity, and here DHT-lookup is only incurred in DHT-get. Note that each DHT-get (in line 6) corresponds to a distinct $f_n(x)$, that is an element in $f_n(\Gamma)$. Since the cardinality of $f_n(\Gamma)$ is approximately $\frac{D}{2}$, the complexity of an LHT lookup operation is $\log(\|f_n(\Gamma(\delta, D))\|) \approx \log(\frac{D}{2})$ DHT lookups. Frankly speaking, our binary search strategy is largely inspired by PHT. However PHT's binary search is simply on Γ and the complexity can be as high as $\log D$. The saving ratio of LHT lookup is $\frac{1}{\log D}$.

Data Insertion and Exact-match Queries LHT lookup can be directly applied to the data insertion and exact-match queries in LHT. The insertion operation involves a LHT lookup of the given data key δ, and after obtaining bucket name κ, a DHT-put towards κ. An LHT insertion may further cause the leaf split. To avoid the cascading split, each insertion in LHT is restricted to cause at most one split. As for the exact-match queries, it is almost equal to an LHT lookup, except that it returns the data record associated with queried δ, rather than the bucket itself.

6 Range Queries

Given two bounds, l and u, a range query returns all records whose keys δ fall between these two bounds, i.e., $\delta \in [l, u)$. Thanks to the local tree, LHT supports near-optimal range query processing. In order to illustrate how it runs, we start from a simple case.

6.1 A Simple Case

In this case, the query initiator happens to be the leaf bucket containing one of the range bound. Without loss of generality, we assume the lower bound l. As mentioned above, the leaf bucket can construct a local tree, as depicted in Fig. 5a. This figure shows all right neighboring trees, denoted as τ_1, τ_2, etc., and neglects the left ones. The τ_i covers the interval $[pv_i, pv_{i+1})$, where partition value pv_i is the lower bound of interval covered by τ_i. The right branch nodes are denoted in the figure as β_1, β_2, etc. In order to show how to precisely infer the right branch node β_i based on the sole knowledge of $\lambda(l)$, we define the *right neighbor function* $f_{rn}(x)$.

Definition 3. *For a tree node labeled with x, the* ***right neighbor function*** *$f_{rn}(x)$ returns the label of its nearest right branch node. For example in Fig. 5a, $f_{rn}(\lambda) = \beta_1$, $f_{rn}(\beta_i) = \beta_{i+1}$.*

$$f_{rn}(x) = \begin{cases} p1 & \textit{if } x{=}p01* \textit{ and } p \neq \#, \\ x & \textit{if } x{=}\#01*. \end{cases}$$

In the case that $x = \#01*$, the tree node x already lies the rightmost in LHT and $f_{rn}(x)$ maps it to itself. We can similarly define the ***left neighbor function*** *$f_{ln}(x)$.*

$$f_{ln}(x) = \begin{cases} p0 & \textit{if } x{=}p10*, \\ x & \textit{if } x{=}\#00*. \end{cases}$$

Using $f_{rn}(x)$, the leaf bucket $\lambda(l)$ can locally infer all β_is, like we will see in the query algorithm. The queried range $[l, u)$ bounds the rightmost branch node β_k, whose neighboring tree τ_k covers the upper bound u, as depicted in Fig. 5a. In order to traverse all leaves in $[l, u)$, leaf bucket $\lambda(l)$ forwards it to the rightmost leaf in β_i for $i = 1, 2...k-1$ and the leftmost leaf in β_k. The process is demonstrated by the arrows in Fig. 5a. As a matter of fact, the forwarding to the rightmost leaf in β_i is done by a DHT-lookup of $f_n(\beta_i)$, because the rightmost leaf in τ_i is named to $f_n(\beta_i)$; and the forwarding to the leftmost leaf in β_k is done by a DHT-lookup of β_k, because the leftmost leaf in τ_k is named to β_k. Current range $[l, u)$ is decomposed into disjoint subranges for the next-hop leaves, specifically, $[pv_i, pv_{i+1})$ for the rightmost leaf in β_i ($i = 1, 2...k-1$); and $[pv_k, u)$, for the leftmost leaf in β_k. By this means, the next leaves still contain one bound of their subrange, i.e., the simple case recursively holds true, and thus the same forwarding strategy applies. This procedure may involve the leaf forwarding from right to left, which is done by similarly applying the f_{ln} to local tree.

For complexity analysis, one point noteworthy here is that during the whole recursive procedure, atmost one DHT-lookup could possibly fail. That is, on the forwarding to the leftmost leaf in τ_k, the β_k may be a leaf node, which leads to a failed DHT-lookup. If that happens, the query is forwarded to $f_n(\beta_k)$ and with no further forwarding needed, the whole procedure is terminated.

Algorithm 3 formally describes the recursive forwarding strategy for the simple case. We brief the flow: It firstly check the direction of the forwarding and infer corresponding branch nodes with the neighbor function (line 1–7); In a loop, it forwards query to $f_n(\beta_i)$ with $i = 1, 2...k-1$ (line 10) and finally to β_k (line 13). As explained, the forwarding to β_k may be failed and is further adjusted towards $f_n(\beta_k)$ (line 17).

Algorithm 3 recursive-forward(bucket b, range R)

```
1:  $leftwards \leftarrow (b.leaflabel = p011*)$
2:  $\beta \leftarrow b.leaflabel$
3:  loop
4:      if $leftwards = true$ then
5:          $\beta \leftarrow f_{ln}(\beta)$
6:      else
7:          $\beta \leftarrow f_{rn}(\beta)$
8:      $inv \leftarrow$ interval($\beta$)
9:      /*compute the interval covered by branch node $\beta$*/
10:     if $inv \subseteq R$ then
11:         $nextbucket \leftarrow$ DHT-lookup($f_n(\beta)$)
12:         recursive-forward($nextbucket$, $inv$)
13:     else
14:         $nextbucket \leftarrow$ DHT-lookup($\beta$)
15:         if $nextbucket = NULL$ then
16:             /*a failed DHT-lookup*/
17:             $nextbucket \leftarrow$ DHT-lookup($f_n(\beta)$)
18:         recursive-forward($nextbucket$, $inv \cap R$)
19:         return
```

Algorithm 4 general-forward(range R)

```
$LCA \leftarrow$ computeLCA($R$).
$bucket \leftarrow$ DHT-lookup($f_n(LCA)$)
if $bucket = NULL$ then
    /*a failed DHT-lookup*/
    return LHT-lookup($R.lowerbound$)
else
    if $bucket$ overlaps $R$ then
        /*turn into the simple case*/
        return recursive-forward($R$, $bucket$)
    else
        $bucket \leftarrow$ DHT-lookup($LCA0$)
        $result0 \leftarrow$ recursive-forward($R \cap bucket.range$, $bucket$)
        $bucket \leftarrow$ DHT-lookup($LCA1$)
        $result1 \leftarrow$ recursive-forward($R \cap bucket.range$, $bucket$)
        return $result0 \cup result1$
```

6.2 General Case

In the general case, the query initiator can be any leaf bucket. The pseudocode is shown in Algorithm 4. After receiving the range query $R = [l, u)$, it locally computes the *lowest common ancestor*, abbreviated as LCA. It then forwards the query by a DHT-lookup of $f_n(LCA)$: Case 1) The DHT-lookup is failed (line 3), implying that the range $[l, u)$ is completely covered by a single leaf. In this case, range processing turns into an exact-match query; Case 2) The returned leaf bucket overlaps the range (line 7), implying one range bound must be in this leaf bucket. So this is the simple case we discussed above. Case 3) The returned leaf bucket doesn't overlap the range (line 10). In this case, LHT further forwards the query via DHT-lookups of $LCA0$ and $LCA1$, which must both turn into the case 2).

An example Consider processing the range query $[0.2, 0.6)$ in the tree of Fig. 5b. Any leaf bucket receiving the query locally calculates the LCA to be $\#0$, and conducts a DHT-lookup of $f_n(\#0) = \#$. The returned leaf bucket is $\#000$ which contains the range lower bound. The recursive forwarding strategy can then apply: it forwards the query to $\#00$ ($= f_n(\#001) = f_n(f_{rn}(\#000))$) and $\#01$ ($= f_{rn}(\#001)$), to which the leaf buckets $\#0011$ and $\#0100$ are respectively named. The bucket $\#0011$ further forwards it to $\#001$ ($= f_n(f_{ln}(\#0011))$) which is the name of bucket $\#0010$. After that, all leaf buckets in the range $[0.2, 0.6)$ are found.

6.3 Complexity

Suppose the range query is distributed on B leaf buckets. We here only consider $B >= 2$ (i.e., the Case 2 and 3). In general forwarding, there is atmost one DHT-lookup which returns a leaf bucket not overlapping the range (i.e., Case 3). As explained, in the procedure of each recursive forwarding, there is atmost one failed DHT-lookup. Therefore, atmost 3 extra DHT-lookups can possibly occur, that is, an LHT range query consumes atmost $B + 3$ DHT-lookups. Noticing that B DHT-lookups are required in the optimal case, we claim the LHT range query algorithm is near-optimal.

7 Min/Max Queries

The min(max) query returns the smallest(largest) data key in a data set. This query type is widely used in various database system. Interestingly, LHT supports efficient processing of a min/max query. Due to the naming function, the complexity is only one DHT-lookup.

Theorem 3. *In LHT, a DHT-lookup of $\#$ returns the result of a min query. Similarly, a DHT-lookup of $\#0$ returns the result of a max query.*

Proof. The leaf bucket containing the smallest data key in LHT should be of the label $\#00*$. By the naming function, this bucket $\#00*$ is mapped to $\#$. Similarly, the largest data key should be stored in leaf bucket $\#01*$, which is named to $\#0$. □

8 Analysis of Tree Maintenance Cost

In this section, we focus on analyzing LHT maintenance cost. Prior to it, we present our cost model which is reasonable for over-DHT indexing schemes.

8.1 Cost Model

Typical P2P network is featured by its abundant resources at the net edges, like local disk storage and CPU computing power. By contrast, the inter-network resource, i.e., the bandwidth is relatively rare, and thus critical. To capture P2P network cost, a simple yet effective way is to consider only bandwidth consumption. For an over-DHT indexing scheme, there are two basic operations that are bandwidth-consuming: the *DHT-lookup* and direct *data-movement* (i.e., transferring data from one peer to any other peer via a physical connection, like TCP or UDP). In this context, we propose a linear cost model, in which moving each data record costs $\imath$ units and each DHT-lookup costs $\jmath$ units. The value of $\imath$ is determined by the size of a data record—for a data record with bigger size, transferring one incurs more bandwidth, yielding a bigger $\imath$. The value of $\jmath$ is determined by the scale of underlying P2P network—for P2P network with more peers, each DHT-lookup incurs more physical hops (typically, at complexity of $O(\log N)$), which leads to a larger $\jmath$.

8.2 Maintenance Cost

Unlike substrate-dependant schemes, LHT has no need of periodical maintenance for index integrality and consistency, for this piece of work is left to and well done by underlying DHT. LHT's maintenance cost is only paid for its tree structure adjustment, incurred by data insertion/deletion. This structural adjustment involves leaf split and merge. Note that they are dual to each other, and for brevity, only leaf split is discussed.

For each leaf split in LHT, only one DHT-lookup is incurred, yielding the DHT-lookup cost of $\jmath$; And the data-movement cost equals the size of remote leaf bucket. Note that for a pair of remote and local buckets, their sizes sum to θ_{split}. Therefore, the remote bucket has its size as a fraction of θ_{split}, denoted as $\alpha \cdot \theta_{split}$, where α is a normalized factor in $[0, 1]$. The local bucket's size is thus $(1 - \alpha) \cdot \theta_{split}$. The very value of α is determined by the local data distribution. For example, with a uniform data distribution, remote bucket equals the local one in size, yielding $\alpha = \frac{1}{2}$. The local data distribution is sensitive to the splitting node, and thus is elusive. However, as we will see in experiments, for the tree large enough, the averaged α (average by all split times in tree growth) approaches $\frac{1}{2}$. Thus, the average data-movement cost per split is $\frac{1}{2}\theta_{split} \cdot \imath$. Altogether, the average cost for a leaf split in LHT, denoted as Ψ_{LHT}, is

$$\Psi_{LHT} = \frac{1}{2}\theta_{split} \cdot \imath + 1 \cdot \jmath \tag{1}$$

In PHT, an index tree similar to space partition tree is maintained, and its mapping to underlying DHT is quite straightforward — All the tree nodes (including the internal nodes) are mapped directly by its label. As a result, one split produces two leaves of changed labels, which are mapped to other peers. These two remote leaf buckets incur data-movement cost of θ_{split} and 2 DHT-lookups. Besides, a split incurs 2 extra DHT-lookups to update its B+ tree leaf links. Altogether, the bandwidth for a PHT split is,

$$\Psi_{PHT} = \theta_{split} \cdot \imath + 4 \cdot \jmath \tag{2}$$

In comparison with PHT, LHT's saving ratio of maintenance cost is,

$$1 - \frac{\Psi_{LHT}}{\Psi_{PHT}} = \frac{\frac{1}{2} \cdot \gamma + 3}{\gamma + 4} \tag{3}$$

where $\gamma = \frac{\theta_{split} \cdot \imath}{\jmath}$. This saving ratio can be up to 75% and at least 50%.

9 Experimental Results

This section presents performance evaluation results of LHT in terms of maintenance cost and query efficiency (including lookup and range query). We compare LHT with PHT, mainly because that PHT is the state-of-the-art indexing scheme with respect to maintenance efficiency.

9.1 Experiment Setup

We implemented LHT in java with only 1700 code lines, which demonstrates the great ease of implementation. For comparison, we also implemented PHT. LHT and PHT were both deployed over Bamboo DHT [18], a ring-like DHT that has good robustness and is now widely deployed in the OpenDHT project [19]. Experiments were conducted in a LAN environment consisting of more than 20 computers (or peers)[5].

Both uniform and gaussian datasets were used. For uniform datasets, the key value is uniformly distributed in $[0, 1]$; and for gaussian datasets, the key value has a gaussian distribution with its mean being $\frac{1}{2}$ and its standard deviation being $\frac{1}{6}$, which guarantees that about 97% key values fall in $[0, 1]$. In each experiment, 100 datasets of each distribution were independently generated, and the averaged results were reported. The size of dataset we used, shortly the *data size*, depends on the value of θ_{split}, which can be as large as 2^{20}. We evaluated both LHT and PHT from three aspects: maintenance cost, LHT lookups performance and range query performance.

9.2 Maintenance Cost

This set of experiments first evaluates the value of average α, and then the maintenance cost.

Average α The average α, as mentioned, is the one averaged by the split times during the whole tree growth. To evaluate it, we continuously inserted data into LHT, and calculated the average α of different data size. Fig. 6a plots the results with the splitting threshold θ_{split} of 40 and 160, respectively. LHT's θ_{split} is then varied, and corresponding results shown in Fig. 6b. Generally, the average α quite approaches $\frac{1}{2}$. Their difference is sensitive to θ_{split}, and for gaussian data, to data size. This is due to that with a small θ_{split}, average α is remarkably affected by extra storage of leaf label. Specifically, in each bucket, like in Fig. 3a, a leaf label occupies one record storage. And a split averagely halves the $\theta_{split} - 1$ "real" records, and assigns one leaf label for each newly produced bucket. Considering this, the average α is,

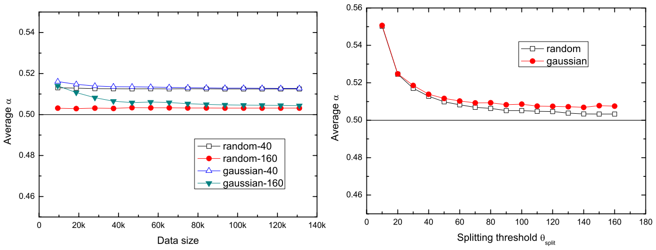
$$\overline{\alpha} = \frac{\frac{\theta-1}{2} + 1}{\theta} = \frac{1}{2} + \frac{1}{2 \cdot \theta}$$

which perfectly matches the plot of uniform data.

Maintenance cost During this experiment, progressively larger dataset is inserted into LHT (as well as PHT), and the cumulative maintenance cost is recorded. In accordance with our cost model, two measurements are used here, i.e., the number of moved records and DHT-lookup numbers. Results of these two measurements are respectively shown in Fig. 7a and Fig. 7b. In Fig. 7a, the θ_{split} is fixed at 100[6], and cumulative data-movement cost basically goes linearly with data size. For uniform data, the curve is fluctuant, and it is owing to the characteristic of binary tree structure. LHT's cost remains half of that of PHT, which is consistent with the value of average α. For DHT-lookups cost, Fig. 7b reveals a similar result, except that LHT's cost is even lower, about 25% of that of PHT.

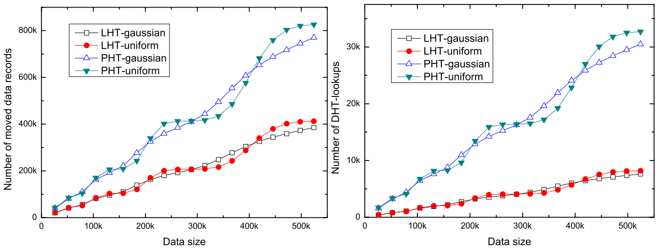
[5] The measurements we used in experiment, like number of DHT-lookups are independent of underlying network scale.

[6] θ_{split} is set to 100 in following experiments, if no explicit announcement is made.



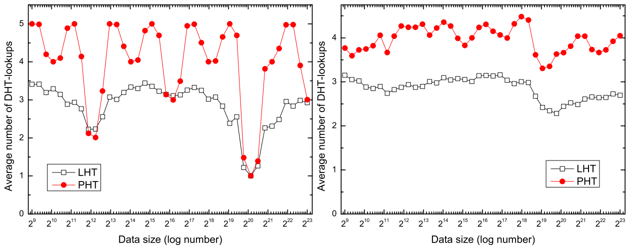
(a) Varying data size (b) Varying θ_{split}

Figure 6: Average α



(a) Data-movement cost (b) DHT-lookup cost

Figure 7: Maintenance cost



(a) With uniform data (b) With gaussian data

Figure 8: Lookup performance

9.3 Lookup Performance

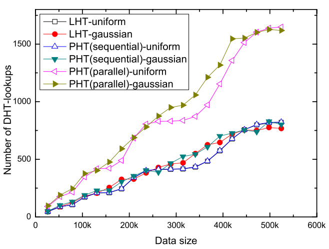
This experiment evaluates lookup efficiency in LHT (and in PHT). Note that the lookup operation has a priori parameter, the maximum possible depth, D, and here it is set to 20. We varied data size, and on each data size, repetitively conducted 1000 lookups for keys that are uniformly distributed in $[0, 1]$. The average DHT-lookup number per lookup operation is reported in Fig. 8. For uniform data in Fig. 8a, two curves are in a peculiar shape, especially for the PHT curve. They both fluctuate with increasing data size, and simultaneously reach some "valley points", like on the data sizes of 2^{12}, 2^{16} and 2^{20}. In these situations, interestingly, the DHT-lookup numbers respectively equal 2, 3, 1. Here is the explanation: when data size equals 2^{20}, leading to a tree of depth of $\log \frac{2^{20}}{100} \simeq 14 = \frac{D}{2}$ (note that data is uniformly distributed), the binary search thus can be resolved in the first try, with only 1 DHT-lookup. For valley points of 2^{12} (2^{16}), similar explanation applies, that the tree depth is $\frac{D}{4}$ ($\frac{3}{8}D$), and binary search resolved in 2 (3) steps. LHT's cost remains lower than PHT's, (except for some valley points), and leads to an average saving ratio of approximate 20%. For gaussian data in Fig. 8b, these two curves are more flat and LHT's saving ratio is roughly 30%.

9.4 Range Query Performance

This set of experiments evaluates LHT's range query performance. Two measurements are considered, that is, bandwidth consumption and time latency. To capture the former, the number of DHT lookups is used, and the data movement cost is here neglected, for each query incurs the same size of data movement (in returning query result). The latter, time latency, is captured by the paralleled steps of DHT lookups, which, unlike the absolute time, is insensitive to the underlying network scale and thus more suitable. During experiments, the query range $[l, u)$ is generated by randomly picking its lower bound l in the interval $[0, 1 - span]$, where the *span* denotes the value of $u - l$. Note that PHT has two range query algorithms, respectively named PHT(sequential) [16], and PHT(parallel) [4]. LHT is here compared to both of them.

The results of bandwidth and latency of range query are shown in Fig. 9 and Fig. 10, respectively. In Fig. 9, for both varied data size and range span, PHT(parallel) incurs the highest bandwidth, while LHT and PHT(sequential) consumes roughly the same bandwidth, which as mentioned, quite approaches the optimal. (Actually, LHT requires slightly less bandwidth, but this difference may not be visible in the figure.) As for time latency, PHT(sequential) is extremely consuming. Note that there are axis breaks in Fig. 10a and Fig. 10b, and in contrast of other two, PHT(sequential) typically requires more time by an order of magnitude. LHT is the most time-efficient, no matter which data type is used. The saving ratio of LHT's latency to PHT(parallel) is approximately 18%. Fig. 10b reveals that LHT's superiority falls down when span goes large with uniform data.

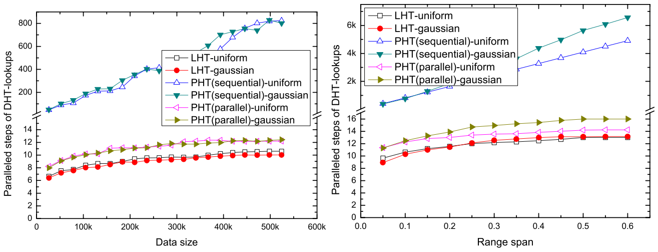
PHT(sequential)'s near-optimal bandwidth consumption is owing to the presence of B+ tree-like leaf link, which on the other hand, incurs extra maintenance cost. Thanks to parallelism, PHT(parallel) can achieve competitive time latency, which however deteriorates when data distribution tends to be skewed (like in gaussian data). Due to nice property of naming function and tree summarization, LHT's



(a) Varying data size

(b) Varying range span

Figure 9: Range query performance (Bandwidth)



(a) Varying data size

(b) Varying range span

Figure 10: Range query performance (Latency)

range query algorithm exceeds in terms of both bandwidth and latency, yet at no expense of maintenance.

10 Conclusion

This paper proposes LHT, a low maintenance hash tree for data indexing over DHTs. As compared with the state-of-the-art indexing structure PHT, LHT can save up to 75% (at least 50%) maintenance cost, and achieves better performance in exact-match and range query processing. This advantage is due to its novel naming function and local tree summarization strategy. LHT is adaptable to any generic DHT, and is easy to be implemented and deployed.

11 Acknowledgement

We thank Shixi Chen and Dr. Jianliang Xu for numerous helpful discussions, and the anonymous referees for their valuable comments. This work is supported by National Natural Science Foundation of China under grants no. 60573183 and no. 90612007; Shuigeng Zhou is also supported by the Shuguang Scholar Program of Shanghai Education Development Foundation.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *SODA*, pages 384–393, 2003.

[2] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. M. Hellerstein. A case study in building layered dht applications. In *SIGCOMM*, pages 97–108, 2005.

[5] L. Chen, K. S. Candan, J. Tatemura, D. Agrawal, and D. Cavendish. On overlay schemes to support point-in-range queries for scalable grid resource discovery. In *P2P Computing*, pages 23–30, 2005.

[6] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB*, pages 25–30, 2004.

[7] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD Conference*, pages 223–234, 2007.

[8] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *P2P Computing*, pages 57–66, 2005.

[9] J. Gao and P. Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *ICNP*, pages 239–250, 2004.

[10] J. Gao and P. Steenkiste. Efficient support for similarity searches in dht-based peer-to-peer systems. In *ICC*, pages 1867–1874, 2007.

[11] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[12] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[13] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *ICDE*, page 34, 2006.

[14] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[15] D. Li, X. Lu, B. Wang, J. Su, J. Cao, K. C. C. Chan, and H. V. Leong. Delay-bounded range queries in dht-based peer-to-peer systems. In *ICDCS*, page 64, 2006.

[16] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. In *PODC*, page 368, 2004.

[17] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[18] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht (awarded best paper!). In *USENIX Annual Technical Conference, General Track*, pages 127–140, 2004.

[19] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *SIGCOMM*, pages 73–84, 2005.

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

[21] O. D. Sahin, A. Gulbeden, F. Emekçi, D. Agrawal, and A. E. Abbadi. Prism: indexing multi-dimensional data in p2p networks using reference vectors. In *ACM Multimedia*, pages 946–955, 2005.

[22] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[23] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *Computer Communication Review*, 32(1):81, 2002.

[24] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over dht. In *The 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2006.