A Lightweight Multidimensional Index for Complex Queries over DHTs

Yuzhe Tang, Jianliang Xu, *Senior Member, IEEE*, Shuigeng Zhou, *Member, IEEE*, Wang-Chien Lee, *Member, IEEE*, Dingxiong Deng, and Yue Wang

Abstract—In this paper, we study the problem of indexing multidimensional data in P2P networks based on distributed hash tables (DHTs). We advocate the indexing approach that superimposes a multidimensional index tree on top of a DHT—a paradigm that keeps the underlying DHT intact while being able to adapt to any DHT substrate. In this context, we identify several index design issues and propose a novel indexing scheme called multidimensional Lightweight Hash Tree (m-LIGHT). First, to preserve data locality, m-LIGHT employs a clever naming mechanism that gracefully maps a tree-based index into the DHT and contributes to high efficiency in both index maintenance and query processing. Second, to tackle the load balancing issue, m-LIGHT leverages a new data-aware splitting strategy that achieves optimal load balance under a fixed index size. We present detailed algorithms for processing complex queries over the m-LIGHT index. We also conduct an extensive performance evaluation of m-LIGHT in comparison with several state-of-the-art indexing schemes. The experimental results show that m-LIGHT substantially reduces index maintenance overhead and improves query performance in terms of both bandwidth consumption and response latency.

Index Terms—P2P systems, distributed hash tables, multi-dimensional indexing, range queries, k-NN queries.

1 Introduction

Distributed Hash Table (DHT) provides a scalable, load balanced, and robust substrate in building large-scale distributed applications. Based on consistent hashing [10], DHT couples data and peers in a unified identifier space. Several DHT overlays, such as Chord [20], CAN [13], and Pastry [17], have been proposed. Although these DHTs employ different identifier spaces and topologies, they share a generic lookup/put/get interface. Specifically, given a key, DHT-lookup locates the peer that stores the key, and DHT-put/DHT-get transfers the associated data to/from the peer located by DHT-lookup.

Whereas simple lookup operations can be efficiently executed over DHTs, there is a lack of support for complex queries, such as range queries and k-nearest neighbor (k-NN) similarity queries, which are however popular in many P2P applications (e.g., "finding the songs that are rated above four and released this year," "finding the top-5 songs with ratings and release dates closest to that of a sample"). The reason is that data locality, which is crucial to processing such complex queries, is destroyed by uniform hashing employed in DHTs.

In the literature, there are two indexing approaches to support complex queries in P2P systems: 1) *over-DHT indexing*, which builds an additional indexing layer on top of generic DHTs (e.g., Prefix Hash Trie (PHT) [5] and Distributed Segment Tree (DST) [25]); 2) *in-DHT indexing*, which modifies the internal structures of underlying DHTs or develops new locality-preserved overlays (e.g., Skip graphs [2] and BATON [8]). Although the over-DHT indexing approach is generally less efficient in query performance than the in-DHT indexing approach, it excels in many other aspects, such as simplicity of deployment/implementation/maintenance, and inherited load balancing [5], [12], [7], [25]. In practice, these issues are equally important to query performance. The over-DHT indexing approach is particularly favorable to the applications in which concerns about ease of implementation, deployment and maintenance dominates the need for high performance, e.g., deploying P2P applications in the world-wide OpenDHT project [15]. In this paper, we study the problem of how to efficiently support multidimensional complex queries in *existing* DHT-based P2P systems and advocate the over-DHT indexing paradigm.

A naive scheme for over-DHT indexing is to simply build a conventional index (e.g., B-tree) on top of a DHT. However, similar to a centralized tree index, processing a query on such an index always requires a traversal from the tree root to leaf nodes (where data are stored). Obviously, this root-to-leaf traversal does not fit well with large-scale P2P systems since the root may easily become the bottleneck. To address this issue, space partitioning has been adopted in the index design [12], [5], [23]. Specifically, a tree-based index partitions the global data space into cells, with each corresponding to a leaf node. By evenly partitioning each data space, the index renders the local

- Y. Tang is with the College of Computing, Georgia Institute of Technology, 329142 Georgia Tech Station, Atlanta, GA 30332. E-mail: yztang@gatech.edu.
- J. Xu is with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong. E-mail: xujl@comp.hkbu.edu.hk.
- S. Zhou is with the Shanghai Key Lab of Intelligent Infromation Processing, School of Computer Science, Fudan University, 220 Handan Road, Shanghai 200433, China. E-mail: sgzhou@fudan.edu.cn.
- W.-C. Lee is with the Department of Computer Science and Engineering, The Pennsylvania State University, 360D Information Sciences and Technology Building, University Park, PA 16802. E-mail: wlee@cse.psu.edu.
- D. Deng and Y. Wang are with the School of Computer Science, Fudan University, 220 Handan Road, Shanghai 200433, China. E-mail: {dxdeng, wangyue}@fudan.edu.cn.

Manuscript received 1 Jan. 2010; revised 16 July 2010; accepted 18 Aug. 2010; published online 11 Mar. 2011.
Recommended for acceptance by M. Singhal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-01-0001.
Digital Object Identifier no. 10.1109/TPDS.2011.91.

1045-9219/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

space indexed by each node to be known globally. Thus, to process a range query, one can start from the lowest internal node that fully covers the query range, instead of always traversing from the root.

In this paper, we follow the space partitioning method and develop a novel index called multidimensional Lightweight Hash Tree (m-LIGHT) over DHTs. In particular, we investigate the following issues: 1) how to map a tree-based index into the DHT for efficiently supporting distributed query processing; and 2) how to perform index maintenance while balancing peer loads. We focus on indexing multidimensional data and employ the *kd-tree* as the basic index in our study. To materialize this index in a distributed setting, we propose a *tree-decomposition* strategy, which enlarges the local view of each peer node, yet requires no extra maintenance overhead. Moreover, we propose a novel multidimensional naming mechanism to map the decomposed tree into the DHT. Our naming mechanism exhibits several nice properties, which lead to high efficiency in both index maintenance and query processing. The contributions of our study are summarized as follows:

- We propose m-LIGHT, a multidimensional data indexing scheme over DHTs, to address both query efficiency and maintenance efficiency. Specifically, we propose a tree-decomposition strategy and a novel naming mechanism to map a kd-tree index into the DHT.
- We develop algorithms for incremental maintenance of the m-LIGHT index. Moreover, we propose a data-aware splitting strategy for load balancing. Given a fixed index size, this strategy achieves the optimal balance of data storage on peer nodes.
- We present algorithms for lookup operations, range queries and k-NN queries over the m-LIGHT index. In particular, we propose a parallel range query algorithm, which provides flexibility between optimizing bandwidth consumption and query response latency.
- We conduct extensive experiments to evaluate m-LIGHT. Compared with the state-of-the-art over-DHT indexing schemes, namely, PHT [5] and DST [25], m-LIGHT substantially reduces index maintenance overhead and strikes a better balance between bandwidth consumption and response latency for complex queries.

The rest of this paper proceeds as follows: Section 2 reviews the related work and presents some preliminaries of over-DHT indexing. Section 3 presents the m-LIGHT index structure, while its lookup operation is described in Section 4. Sections 5 and 6 give the algorithms for processing range queries and k-NN queries, respectively. How to update the m-LIGHT index is presented in Section 7. Section 8 experimentally evaluates the performance of m-LIGHT. Finally, Section 9 concludes this paper.

2 Preliminaries and Related Work

Before we present the proposed m-LIGHT index, in this section, we give a brief introduction to 1D over-DHT indexes and review the related work.

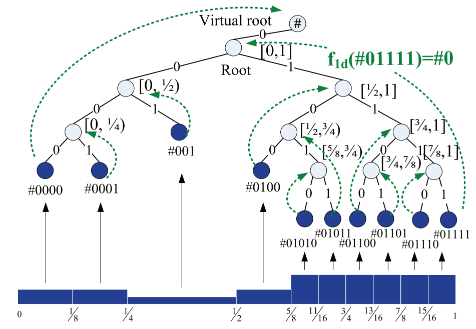


Fig. 1. Binary tree and naming schemes.

2.1 Over-DHT Indexes for 1D Data

Consider a set of *data records*, each of which is identified by a *data key*, denoted by δ ($0 \leq \delta < 1$). The data records are assigned to the underlying DHT based on their data keys, i.e., using *DHT-put(δ)*. Thus, exact-match queries can be directly supported by the DHT using *DHT-get(δ)*. On the other hand, to expedite complex query processing, a binary index tree is constructed on top of the DHT. It recursively partitions the data space into two equal-sized subspace until the number of data records covered by each subspace is less than a predefined threshold (see Fig. 1 for an example). In the index tree, leaf nodes store pointers pointing to actual data records.

A key issue for over-DHT indexes is how to store and maintain the index nodes in the distributed DHT network. A variety of solutions have been proposed. In a pioneering work, PHT [5], each edge is labeled, so does each index node. Specifically, if an edge connects to the left child of an internal node, it is labeled with bit 0, otherwise 1. Each node is labeled with a binary string that concatenates all bits along the path from the root to the node. Then, the label of each index node is used as its *DHT key*, based on which the index node is distributed in the DHT network. For example, in Fig. 1, the leftmost leaf node is labeled with #0000 and stored in the network based on the DHT key #0000. To process a range query, PHT figures out the lowest common ancestor (LCA) that fully covers the query range. For example, given a query range [0.1, 0.3], the LCA is the node labeled with #00. It then locates this node and starting from there traverses all the way down to the leaf nodes overlapping with the query range. The internal nodes in PHT do not hold data and serve as routing nodes only. To further improve query performance, DST [25] and Range Search Tree (RST) [7] have been proposed to replicate data in internal nodes. But as a side effect, the index update overhead is increased significantly. In addition, when a leaf node splits, two new child nodes will be generated with new labels; both of them need to be redistributed in the DHT network.

To achieve both query efficiency and index maintenance efficiency, Lightweight Hash Tree (LIGHT) was proposed in our prior work [21], [22]. In LIGHT, node labels are not directly used as DHT keys of index nodes. Instead, DHT keys are generated from node labels using a naming function $f_{1d}(\cdot)$. Specifically, $f_{1d}(\cdot)$ *names* each leaf node to a distinct internal node, as shown by the dashed arrows in Fig. 1. For example, the rightmost leaf #01111 is *named* to the internal node $f_{1d}(\#01111) = \#0$, and is stored in the

DHT based on the DHT key #0. Formally, we have the naming function defined as follows:

$$f_{1d}(b_1 \cdots b_{i-2}b_{i-1}b_i) = \begin{cases} f_{1d}(b_1 \cdots b_{i-2}b_{i-1}), & \text{if } b_{i-1} = b_i, \\ b_1 \cdots b_{i-2}b_{i-1}, & \text{otherwise.} \end{cases}$$

By adopting this naming function $f_{1d}(\cdot)$, only leaf nodes are stored in the DHT network [22]. A range query can be efficiently processed by first locating the leftmost leaf node that satisfies the range query, and then, recursively forwarding the query to its right neighbors. Meanwhile, LIGHT also reduces significant amount of index maintenance overhead. When a leaf node splits, the DHT key of one child will remain unchanged thanks to the nice naming function. Thus, half of data movement is saved. Nevertheless, LIGHT deals with 1D data only.

2.2 Over-DHT Indexes for Multidimensional Data

With 1D over-DHT indexes, there are generally three choices for processing multidimensional queries. The first is to construct multiple independent indexes, with each indexing 1D. A multidimensional query is then processed in parallel over multiple indexes [1], [3], [4]. However, this solution incurs a high query processing and index maintenance overhead. The second choice is Space Filling Curve (SFC) indexing, which uses the SFC to reduce data dimensionality and then indexes data by some 1D index [5], [6], [18]. But the problem in SFC indexing is that the spatial proximity in the multidimensional space may not be preserved in the 1D SFC, which deteriorates query efficiency. To date, the most promising solution is to directly build multidimensional indexes. A typical approach is to index data by conventional multidimensional indexes (e.g., kd-tree) and map these indexes into DHT space, e.g., [6], [9], [11], [24]. Unfortunately, these existing schemes rely on specific P2P networks. In contrast, in this paper, we are interested to develop novel multidimensional indexes based on generic DHTs.

3 m-LIGHT Indexing Scheme

In this section, we follow the design principle of LIGHT [22] and propose a multidimensional indexing scheme, call m-LIGHT. We note that the extension is not straightforward. First, the label-to-DHT-key naming mechanism should be redesigned to preserve the spatial proximity in multidimensional space. Second, processing complex queries in multidimensional space is much more complicated than in 1D space. Third, space partitioning index tree has a known problem of load imbalance. This problem might be aggravated in multidimensional space. Thus, index update strategies should be aware of load distributions of index nodes. In the following, we first introduce the space kd-tree for indexing data in multidimensional space.

3.1 Distributed Indexing of Space Kd-Tree

To index multidimensional data, we recursively partition the data space into *cells* along different dimensions in an alternative fashion. For example, in Fig. 2a, the 2D space is recursively halved along the x and y axes, alternatively, until a cell contains no more than θ_{split} data records, where θ_{split} is a predefined threshold. Space partitioning is used

Fig. 2. Space kd-tree. (a) Space partitioning. (b) Space kd-tree decomposition (with each box corresponding to a local tree).

here, that is, a data space is always equally partitioned, regardless of the data distribution. As mentioned, such a space is essential to distributed query processing. We call the index the *space kd-tree*, in which each leaf corresponds to a distinct cell, and each internal node corresponds to a *region* that covers several cells. As shown in Fig. 2b, every internal node has two children. However, unlike a conventional kd-tree, the space kd-tree has two roots. The additional root, termed as *virtual root*, is a virtual node above the ordinary one. Thus, the number of leaf nodes equals the number of nonleaf nodes. As will be shown later, this property enables us to name each leaf node with a distinct internal node. The labeling strategy is similar to that in the 1D case. If the tree edge goes left, it is labeled with 0; otherwise 1. In particular, in the 2D case, the label of the virtual root is 00 and the label of the ordinary root is 001 (denoted by # hereafter, $\# = 001$). Thus, for example, the leftmost leaf node in Fig. 2b is labeled with #000. This is consistent with the one visualized in Fig. 2a, where 0/1 represents the cell that is near/apart from the origin in each space partition, and the label of a cell is obtained by interleaving the labels for all the partitions along the x and y axes. Since each label is unique, it can be used to denote the corresponding node in the space kd-tree.

To materialize the tree in a distributed setting, we decompose the space kd-tree and store each piece into a *leaf bucket*. Conceptually, a leaf bucket is a *local tree* consisting of a distinct leaf node and all its ancestors. For example, in Fig. 2b, the two bounding boxes illustrate two local trees of leaves #01 and #101111.

3.2 m-Dimensional Naming Function

For a leaf bucket labeled with λ, the m-dimensional naming function $f_{md}(\cdot)$ generates its DHT key κ, i.e., $\kappa = f_{md}(\lambda)$. The bucket is then stored on the peer node based on the DHT key. In this section, we first present the naming function for 2D indexing, and then extend it to m-dimensional indexing.

3.2.1 Naming for 2D Indexing

Definition 1 (2D-Naming Function). *In a 2D space kd-tree, given the binary label of any leaf,* $\lambda = b_1 \cdots b_{i-2}b_{i-1}b_i$, *where* $b_j = [0|1] (j = 1, \ldots, i)$, *the* 2D-naming function *is recursively defined as follows:*

$$f_{2d}(b_1 \cdots b_{i-2}b_{i-1}b_i) = \begin{cases} f_{2d}(b_1 \cdots b_{i-2}b_{i-1}), & \text{if } b_{i-2} = b_i, \\ b_1 \cdots b_{i-2}b_{i-1}, & \text{otherwise.} \end{cases}$$

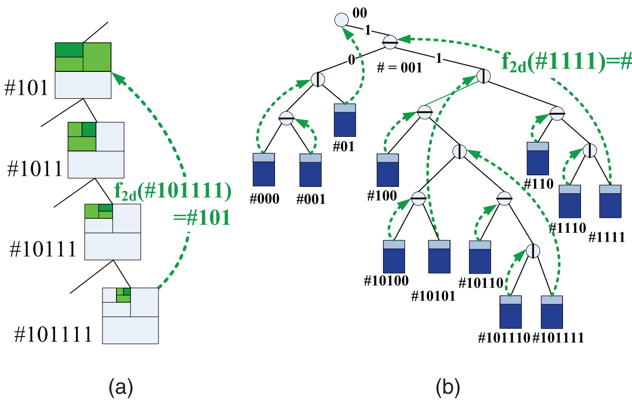


Fig. 3. Naming the space kd-tree in m-LIGHT. (a) The naming function. (b) Naming bijectively from leaves to internal nodes.

Specifically, given a binary string $\lambda = b_1 \cdots b_{i-2}b_{i-1}b_i$, $f_{2d}(\cdot)$ checks its last bit b_i and the last third bit b_{i-2}. If they are the same, the last bit is truncated and this procedure is repeated. Otherwise, the procedure is terminated after truncating the last bit. Thus, $f_{2d}(\lambda)$ always produces a prefix of λ. For example, $f_{2d}(\#0101111) = \#0101$, $f_{2d}(\#0011111) = \#001$, and $f_{2d}(\#101111) = \#101$, as in Fig. 3. In particular, $f_{2d}(\#) = f_{2d}(001) = 00$.

Next, we show that the naming function $f_{2d}(\cdot)$ has two important properties, including *corner preservation* and *bijectiveness*.[1]

Theorem 1 (Corner Preservation). *Given an internal node ω, the corresponding data region has four corners, and the cells lying on them are, respectively, named to $f_{2d}(\omega), \omega, \omega 0$, and $\omega 1$.*[2]

Theorem 2 (Bijective Mapping). *Let Λ and Ω denote the sets of leaves and internal nodes, respectively. $f_{2d}(\cdot)$ is a bijective mapping from Λ to Ω.*

Theorem 1 implies that given ω, the names of its four corner cells can be directly inferred, which, as will be shown later, is very useful for distributed range queries (since it helps to quickly locate the range boundaries). Theorem 2 guarantees that for each DHT key (i.e., the label of an internal node), there is one and only one leaf named to it.

3.2.2 Scale up to m-Dimensional Indexing

Definition 2 (m-Dimensional Naming Function). *In a space kd-tree, given any leaf label $\lambda = b_1 \cdots b_{i-m} \cdots b_{i-1}b_i$, where $b_j = [0|1](j = 1, \ldots, i)$, the* m-dimensional naming function *is recursively defined by*

$$f_{md}(\lambda) = f_{md}(b_1 \cdots b_{i-m} \cdots b_{i-1}b_i),$$
$$= \begin{cases} f_{md}(b_1 \cdots b_{i-m} \cdots b_{i-1}), & if\ b_{i-m} = b_i, \\ b_1 \cdots b_{i-m} \cdots b_{i-1}, & otherwise. \end{cases}$$

In the general case, the root label $\#$ is adjusted to be $0 \cdots 01$ (with m consecutive 0's). Then, the theorems presented in the previous section also hold for m-dimensional space.

1. The proofs of all theorems are given in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91.

2. We assume the subtree rooted at ω is deep enough that all four corner cells exist. For the case where ω has only two leaves (or cells), they are named to $f_{2d}(\omega)$ and ω.

Theorem 3 (m-Dimensional Corner Preservation). *Given any internal node ω in the m-dimensional index tree, the corresponding data cube has 2^m corners, and the cells lying on them are, respectively, named to $f_{md}(\omega), \omega, \omega 0, \omega 1, \omega 00, \omega 01, \ldots$, and $\omega 11 \cdots 1$ (note that the last name has $m-1$ consecutive 1's in the end).*

Theorem 4 (m-Dimensional Bijective Mapping). *$f_{md}(\cdot)$ is a bijective mapping from Λ to Ω.*

In the rest of this paper, for simplicity our discussions will be mainly based on 2D indexing. Nevertheless, all algorithms and proofs can be extended to m-dimensional indexing in a natural way.

4 Lookup Operation

Given a data key δ, the m-LIGHT lookup operation[3] returns the label of the leaf bucket that covers δ, namely, $\lambda(\delta)$. The lookup operation serves as a basis for complex query processing.

Recall that m-LIGHT leverages space partitioning. Any peer receiving a lookup operation for δ can locally calculate the set of all possible values of $\lambda(\delta)$, which is called the candidate set. To be more precise, for a 2D data key $\delta = <\delta_1, \delta_2>$, one can get the binary representations for δ_1 and δ_2, respectively, and then interleave them. For example, given $\delta = <0.2, 0.4>$, the binary representations for 0.2 and 0.4 are $001\cdots$ and $011\ldots$, respectively. Then, the interleaved binary number is $001011\ldots$, and the target label $\lambda(<0.2, 0.4>)$ must be a prefix of $\#001011\ldots$. In Fig. 2a, $\lambda(<0.2, 0.4>) = \#001$.

To find the target label $\lambda(\delta)$, we start with discovering the possible maximum length of the label. Given a predefined guess on the maximum length, D', we conduct a DHT-lookup for the prefix of the label of length D'. If it fails (i.e., returning a $NULL$ value), we know that the maximum length must be less than D'. Otherwise, we will proceed to probe $2D', 4D', 8D', \ldots$, until a failed DHT-lookup, say with depth $2^dD'$, is encountered. Then, the length of the target label is known to be in interval $(2^{d-1}D', 2^dD')$. In general, after the discovery phase, we know that the target label has a length in a certain range, denoted by $(s, s+D)$.

Then, we employ binary search to find the exact length of the target label. Specifically, in each loop iteration, the algorithm probes the candidate label with the length being the middle value of an interval, which is initialized as $[s, s+D)$. However, unlike the basic binary search, due to our unique naming function, the m-LIGHT lookup can perform some pruning during the search process. That is, when the probe for candidate label λ_c fails, implying λ_c is already deeper than the target leaf, the upper bound of the interval is thus set to $f_{2d}(\lambda_c)$ (rather than λ_c). This is because in this probe, m-LIGHT conducts a DHT-lookup for $f_{2d}(\lambda_c)$, and it actually has examined all labels between $f_{2d}(\lambda_c)$ and λ_c, since they are all named to $f_{2d}(\lambda_c)$. Similarly, if the current probe returns a bucket that does not cover δ, implying the label tried is too short, the interval's lower

3. In this paper, we refer to m-LIGHT lookup as "lookup" for short, and for clarity the DHT-lookup remains its full name.

bound is then adjusted to $lnx(f_{2d}(\lambda_c))$ (rather than λ_c), where $lnx(f_{2d}(\lambda_c))$ denotes the longest candidate label named to $f_{2d}(\lambda_c)$. This search process continues until it returns a leaf bucket that covers δ. A lookup example, and the complexity analysis are presented in Appendix B.1 and Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91, respectively.

5 Range Queries

In a multidimensional space, a range query specifies a multidimensional region and returns all data keys falling in that region. In this section, we present the range query algorithm over the m-LIGHT index, where the queried region can be of an arbitrary shape.

5.1 Basic Range Query Algorithm

Consider a query with range R. Upon receiving the query from the user, the peer (i.e., *query initiator*) can locally figure out the lowest internal node that fully covers R (i.e., the LCA of R). The algorithm then proceeds to forward the range query to the LCA. Specifically, the query initiator carries out a DHT-lookup of $f_{2d}(LCA)$, which must reach one corner cell in the region associated with the LCA, as shown in Theorem 1. Upon receiving the range query, the corner cell constructs a local tree based on its leaf label. Among all *branch nodes* (i.e., the second children of the ancestors) in the local tree, there exist one or more whose regions overlap the query range. Denote these branch nodes by $\beta_1, \beta_2, \ldots,$ and β_l, respectively. For each β_i, the range query is decomposed into the subrange R_i, which is the overlapped region between β_i and R, i.e., $R_i = \beta_i \cap R$. Then, R_i is forwarded to β_i via a DHT-lookup of $f_{2d}(\beta_i)$. Note that there is no overlap between R_i and R_j due to the space partitioning method employed in m-LIGHT. Hence, the subqueries $R_i (i = 1, 2, \ldots, l)$ can be processed in parallel and no bucket revisit is needed. For further forwarding in each β_i, a similar process can be recursively applied until the current query range is fully covered in one cell.

Algorithms 1 and 2 formally describe the range query processing with m-LIGHT. The query initiator executes Algorithm 1. It probes the name of the LCA of the query range by a DHT-lookup, and there are three cases:

1. The DHT-lookup fails, which implies the query range is so small that a single leaf bucket can cover it. Thus, the range query is reduced to a lookup query.
2. The located bucket fully covers the query range; so the range query is directly resolved.
3. Otherwise, the LCA contains more than one cell, and the range query needs to be further processed. In this case, it applies Algorithm 2, which is detailed in the last paragraph.

Algorithm 1. range-query(range R)
1: $\omega_R \leftarrow$ lowest-common-ancestor(R)
2: $\lambda \leftarrow$ DHT-lookup($f_{md}(\omega_R)$)
3: **if** $\lambda == NULL$ **then**
4: **return** lookup(R.top_left_corner)
5: **else if** $R \subseteq \lambda$ **then**
6: **return** the keys of λ that are in the range R
7: **else**
8: **return** recursive-forward(R, ω_R)

Algorithm 2. recursive-forward(range R, region β)
1: $\lambda \leftarrow$ DHT-lookup($f_{md}(\beta)$)
2: **for all** $\beta_i \in \{\text{branch nodes between } \lambda \text{ and } \beta\}$ **do**
3: $R_i \leftarrow \beta_i \cap R$
4: **if** $R_i \neq NULL$ **then**
5: recursive-forward(R_i, β_i)

5.2 Parallel Range Query Algorithm

In many distributed query algorithms, there exists a tradeoff between query bandwidth and response latency. Inspired by this observation, we propose a parallel query algorithm to provide such flexibility.

The basic idea is to simultaneously forward two subqueries (rather than one) within a branch node. Specifically, each branch node β represents a space region; and from Theorem 1, region β has two corner cells that are, respectively, named to $f_{2d}(\beta)$ and β (denoted by λ_1 and λ_2). To forward the range query, we now perform two DHT-lookups to both $f_{2d}(\beta)$ and β, which deliver the query to λ_1 and λ_2. These two DHT-lookups run in parallel and, hence, each recursive forwarding can explore the neighboring subtree by two levels (instead of just one). As a result, the total latency can be reduced approximately by a factor of two.

To further improve the bandwidth efficiency, we refine the parallel process by taking the range location into account. Specifically, the branch node β may be further partitioned into two halves, covering λ_1 and λ_2, respectively. If the query range R is completely contained in the partition of λ_1, the query is then forwarded to λ_1 only. Otherwise, the query is forwarded to both λ_1 and λ_2.

The formal procedure is described in Algorithms 3. In general, if we forward h steps ahead within a branch node, the average latency can be reduced approximately by a factor of $h + 1$, but with the number of DHT-lookups also increased. In practice, the user can tune the parameter of h based on his/her performance preference. Appendix B.2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91, gives a basic range query example and a parallel range query example.

Algorithm 3. parallel-recursive-forward(range R, region β, lookahead steps h)
1: $\lambda \leftarrow$ DHT-lookup($f_{md}(\beta)$)
2: **for all** $\beta_i \in \{\text{branch nodes between } \lambda \text{ and } \beta\}$ **do**
3: **for all** $\beta_{i,j} \in \{\text{leaves of subtree rooted at } \beta_i,$ $\text{of depth } h\}$ **do**
4: $R_{i,j} \leftarrow \beta_{i,j} \cap R$
5: **if** $R_{i,j} \neq NULL$ **then**
6: parallel-recursive-forward($R_{i,j}$, $\beta_{i,j}$, h)

6 k-NN Queries

Given a query point Q and an integer k, a k-NN query returns the k data keys that are nearest to Q. In this section, we present the k-NN query algorithm over the m-LIGHT index.

First, we introduce a notion of MINDIST for measuring the distance between a point and a rectangle [16]. In a space kd-tree, each leaf node corresponds to a distinct cell, and each internal node corresponds to a region that covers a number of cells, all of which are in a rectangular shape. MINDIST(p, R) is the minimum distance between a point p and a rectangle R. If p is inside the rectangle, MINDIST(p, R) is zero; if the point is outside R, MINDIST(p, R) is the euclidean distance between the point p and the nearest point of the rectangle. In our algorithm, if MINDIST(Q, I) between an index node I and the query point Q is greater than the k-NN distance we have found so far, this node I is discarded since all keys contained in I are farther away than the current kth NN.

The k-NN query algorithm works as follows: We start by locating the leaf node that contains the query point Q. Then, like processing range queries, we gradually expand the search space by visiting all branch nodes of the leaf node until all k nearest keys are found. Algorithm 4 gives the formal description of k-NN query processing. The peer first conducts a lookup to find the label of the leaf node that contains Q. Then, it carries out a DHT-lookup for that leaf node and computes a candidate k-NN result set based on the retrieved keys (lines 1-3). After that, all branch nodes of the leaf node are inserted into a priority queue $\mathcal{H}$ according to their MINDIST distances to Q (lines 4-5). Each time while the queue $\mathcal{H}$ is not empty, a node λ is dequeued (line 7). If the candidate results are less than k or MINDIST(Q, λ) is shorter than the kth distance—the distance of the kth NN found so far, the algorithm proceeds to check the status of the node λ. If λ is a leaf node, we perform a DHT-lookup of $f_{md}(\lambda)$, update the candidate result set and the kth distance (lines 10-12). Otherwise, if λ is an internal node, a DHT-lookup of $f_{md}(\lambda)$ is performed, which will reach one corner cell γ of the space λ. For γ and each branch node β_i between γ and λ, they will be inserted into the priority queue $\mathcal{H}$ (lines 14-17). Appendix B.3, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91, gives an example of k-NN query.

Algorithm 4. k-NN-query(query point Q, k)

```
1:  β ← Lookup(Q)
2:  DHT-lookup(f_md(β))
3:  Retrieve the keys of the bucket and compute a
    candidate k-NN result set C
4:  for all β_i ∈ {branch nodes between β and root} do
5:      Enqueue(H, β_i)
6:  while H is not empty do
7:      λ ← dequeue(H)
8:      if (|C| < k or (MINDIST(Q, λ) < kth distance) then
9:          DHT-lookup(λ)
10:         if λ is a leaf node then
11:             Retrieve the keys of the bucket
12:             Update the candidate result set C and kth
                distance
13:         else
14:             γ ← DHT-lookup(f_md(λ))
15:             Enqueue(H, γ)
16:             for all β_i ∈ {branch nodes between γ and λ} do
17:                 Enqueue(H, β_i)
```

7 Index Tree Maintenance

In this section, we discuss how m-LIGHT adjusts its structure along with data insertions and deletions.[4] We first consider the conventional threshold-based splitting strategy and show that m-LIGHT can achieve *incremental tree maintenance*. After that, we propose a data-aware splitting strategy which offers optimal load balance among leaf buckets. The data deletion algorithm is presented in Appendix D.3, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91.

7.1 Incremental Tree Maintenance

To adapt the tree structure to current data distribution, we follow the conventional threshold-based splitting strategy. That is, we predefine two thresholds, namely, θ_{split} and θ_{merge}, for leaf split and merge. After a data insertion, if the number of records stored in a leaf bucket gets higher than θ_{split}, a split process is triggered. Similarly, after a data deletion, a pair of sibling leaf buckets may be found containing less than θ_{merge} data records, and a leaf merge is thus triggered. Here, for simplicity, we restrict that for each data insertion/deletion operation, only one leaf split/merge is allowed (i.e., no cascade split/merge).

Now we are going to show how the split/merge process actually works under this framework. First, we present a property of the 2D naming function.

Theorem 5 (Incremental Split). *Consider a leaf, say λ, is split into two nodes. The naming function $f_{2d}(\cdot)$ maps one to $f_{2d}(\lambda)$, and the other to λ.*

The split process proceeds as follows: The splitting bucket λ is first divided into two buckets locally. Then, it conducts a DHT-put operation to reassign the bucket named to λ in the underlying DHT. For the other one named to $f_{2d}(\lambda)$, it shares the same DHT key with the splitting bucket λ; thus, it is mapped to the same peer and no data migration is needed. Similarly, to merge a pair of leaf buckets, only one bucket needs to be transferred across the DHT. Hence, among two operating buckets in the split/merge, there is always one retained on the previous peer. This nice property, termed as *incremental tree maintenance*, can reduce the cost by half in terms of both the number of DHT-lookups and the amount of transferred data.

7.2 Data-Aware Splitting Strategy

We observe that the threshold-based splitting strategy may generate empty leaf buckets, since it does not take into account the local data distribution when partitioning data space. However, generating empty buckets is against the purpose of the splitting process (i.e., balancing the load among leaf buckets). Here, we propose a data-aware splitting strategy which can achieve optimal load balance among leaf buckets.

The data-aware splitting strategy requires a predefined parameter ϵ, which indicates the expected load (rather than the upper/lower bound) in terms of the number of data records stored in each bucket. Generally speaking, this strategy aims at minimizing the difference between the real

4. In terms of index maintenance, m-LIGHT does not deal with peer failures, which are handled by the underlying DHTs leveraging techniques, like data replication.

load and the expected one (i.e., ϵ). When a bucket receives a new data record, it locally computes a virtual subtree rooted at this bucket, called *optimal split subtree*, which minimizes the total *difference* for all leaves. Specifically, for a leaf bucket, the difference is $(l - \epsilon)^2$, where l is the number of data records stored in the bucket. To work out the minimized total difference, a naive solution is to apply the brutal-force search to try all possibilities, which is however time-consuming. Instead, we use a divide-and-conquer approach, as shown in Algorithm 5—it first computes the minimized total difference for the left child, and then for the right child. The process is recursively invoked until the cell containing no more than ϵ data points is reached (line 2). When the computation is done, we compare the minimized value with the current difference. If the minimized value is smaller, the current bucket is split according to the optimal split subtree; otherwise, it stays unchanged. Note that the algorithm runs locally and is invoked every time a bucket changes its load (due to data insertions/deletions). Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91, shows an example and proves the optimality of this strategy in storage load balance.

Algorithm 5. local-split(leaf bucket λ)
1: $s_{local} \leftarrow (\lambda.load - \epsilon)^2$
2: **if** λ.load $\leq \epsilon$ **then**
3: **return** s_{local}.
4: **else**
5: $s_{left} \leftarrow$ local-split(λ.leftChild())
6: $s_{right} \leftarrow$ local-split(λ.rightChild())
7: $s_{non_local} \leftarrow s_{left} + s_{right}$
8: **if** $s_{local} \leq s_{non_local}$ **then**
9: **return** s_{local}.
10: **else**
11: **return** s_{non_local}.

8 PERFORMANCE EVALUATION

This section presents the results of performance evaluation. Note that m-LIGHT belongs to the multidimensional over-DHT indexing. Thus, we compare it with the state-of-the-art schemes in the same category, that is, PHT [5] and DST [25], [19]. The performance metrics of our interest are index maintenance overhead, load balance, query cost, and their scalability to high dimensionality. Other experiment results including scalability to high dimensionality are shown in Appendix E, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.91.

8.1 Experiment Setup

We have implemented the m-LIGHT index in Java. The total number of code lines is about 2,500 (including m-LIGHT, DST, and PHT), which demonstrates the simplicity of developing an over-DHT indexing scheme. In the experiments, m-LIGHT, DST, and PHT were run over the Bamboo DHT [14], a ring-like DHT that has good robustness and is now deployed in a real-life project, OpenDHT [15]. Our experimental study is based on a system built in an Internet environment.

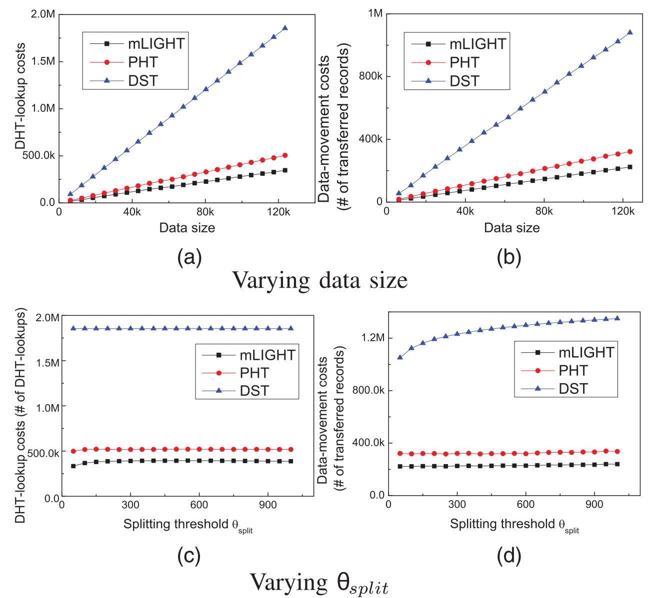


Fig. 4. Maintenance costs. (a) DHT-lookup costs. (b) Data-movement costs. (c) DHT-lookup costs. (d) Data-movement costs.

In the experiments, we first tested real geographic data and then synthetic data for scalability evaluation. The real North-East data set contains 123,593 postal addresses (points) in three metropolitan areas of New York, Philadelphia, and Boston. Along each dimension, we normalize the data points into the range [0, 1]. In the experiments, we inserted these data points progressively into the index, and tested its performance under different data set sizes.

8.2 Index Maintenance Performance

The first experiment evaluates the maintenance performance of m-LIGHT when data are progressively inserted. Recall that data insertion in m-LIGHT involves two operations: a lookup and a possible leaf bucket split. Both of these two operations incur system costs, and in this experiment, we report these costs as a whole. We take two measures, that is, the DHT-lookup cost and the data-movement cost. The results are shown in Figs. 4a and 4b. For all three indexing schemes under comparison, the cumulative maintenance costs go up linearly as more data are inserted. We also vary the threshold θ_{split} and report the evaluation results in Figs. 4c and 4d. In general, both of DHT-lookup cost and data-movement cost are insensitive to the value of θ_{split}, except that DST incurs less data-movement cost when θ_{split} is smaller. This is because in this case, the internal nodes in DST easily get saturated, and many data records are not replicated on these nodes, thereby decreasing the data-movement cost. Comparing the three indexing schemes, due to the replication strategy, DST is much more costly than the other two by an order of magnitude; the m-LIGHT index achieves the best performance in all cases tested and saves about 40 percent maintenance cost against PHT.

8.3 Range Query Performance

We now evaluate the range query performance in terms of bandwidth cost and response latency. The two measures are, respectively, captured by the number of DHT-lookups and the rounds of DHT-lookups. In the evaluation, we include

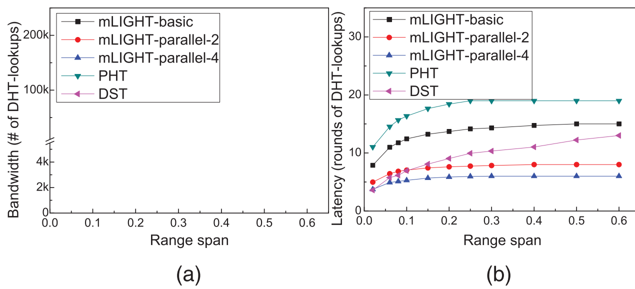


Fig. 5. Range query performance. (a) Bandwidth. (b) Latency.

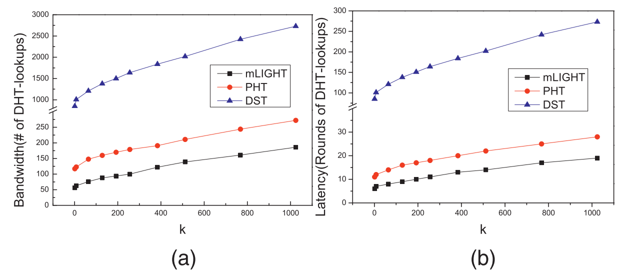


Fig. 6. k-NN query performance. (a) Bandwidth. (b) Latency.

both the basic algorithm and the parallel algorithm for m-LIGHT. For the parallel algorithm, we test two versions, with the parameter of *lookahead steps* being 2 and 4, respectively. We compare the three m-LIGHT query algorithms with PHT and DST. In the experiments, the queried ranges are rectangles uniformly distributed in the data space of $[0\ldots1, 0\ldots1]$. We first vary the range span (i.e., the area of the rectangle) and report the results in Figs. 5a and 5b. In terms of bandwidth cost, DST consumes much more than any other scheme, typically by an order of magnitude. Because in our setting, D' being 28 is much bigger than the real tree depth, rendering the queried range decomposed into many small subranges in DST. In contrast, m-LIGHT (basic) is the most bandwidth-efficient. The m-LIGHT (parallel-2) and m-LIGHT (parallel-4) consume more bandwidth, but as a trade-off, they achieve a significant saving in query latency. DST is time-efficient when the ranges are small. However, as the ranges become larger, the latency of DST dramatically increases, whereas all other schemes are more stable as shown in Fig. 5b.

8.4 k-NN Query Performance

We evaluate the k-NN query performance in terms of bandwidth cost and response latency. In the evaluation, we compare the m-LIGHT query algorithm with PHT and DST. We randomly generate 100 query points for 100 k-NN queries. Figs. 6a and 6b plot the average results of these k-NN queries with k ranging from 1 to 1,024. In general, the query costs go up linearly as k increases. In terms of the bandwidth cost (Fig. 6a), DST consumes much more than the other two schemes, and m-LIGHT is the best one thanks to its nice local tree property. As for the latency, m-LIGHT achieves the best performance among the three schemes as shown in Fig. 6b.

In summary, the proposed m-LIGHT is more flexible and outperforms PHT and DST in many aspects, including index maintenance and query processing. Moreover, m-LIGHT (parallel) trades bandwidth efficiency for significant saving in query latency. DST has a comparable performance with m-LIGHT in terms of range query latency. However, it has an extremely high cost in index maintenance and query bandwidth.

9 Conclusion

This paper has proposed m-LIGHT, an efficient multidimensional index structure for supporting complex query processing over DHTs. Three core techniques contribute to the efficiency m-LIGHT: a tree-decomposition strategy, a novel naming mechanism, and a data-aware splitting strategy. Experimental results based on a real data set show that m-LIGHT outperforms state-of-the-art schemes in various aspects, including maintenance efficiency, lookup performance, and query performance. As an over-DHT indexing scheme, m-LIGHT is able to adapt to any DHT substrate, and is also easy to implement and deploy.

Acknowledgments

The work of Yuezhe Tang, Shuigeng Zhou, Dingxiong Deng, and Yue Wang were supported by the National Basic Research Program of China under Grant No. 2007CB310806, National Natural Science Foundation of China (NSFC) under Grant No. 60873070, and 863 Program under Grant No. 2009AA01Z135. Shuigeng Zhou was also supported by K.C. Wong Education Foundation-HKBU. Jianliang Xu's work was supported by the Research Grants Council of Hong Kong under Projects HKBU211307 and HKBU211510. Wang-Chien Lee was supported in part by US National Science Foundation Grant CNS-0626709 and Grant IIS-0534343.

References

[1] A. Andrzejak and Z. Xu, "Scalable, Efficient Range Queries for Grid Information Services," *Proc. Peer-to-Peer Computing,* pp. 33-40, 2002.

[2] J. Aspnes and G. Shah, "Skip Graphs," *Proc. 14th Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '03),* pp. 384-393, 2003.

[3] A.R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *Proc. ACM SIGCOMM,* pp. 353-366, 2004.

[4] M. Cai, M.R. Frank, J. Chen, and P.A. Szekely, "MAAN: A Multi-Attribute Addressable Network for Grid Information Services," *Proc. Fourth Int'l Workshop Gird Computing (GIRD '03),* pp. 184-191, 2003.

[5] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J.M. Hellerstein, "A Case Study in Building Layered DHT Applications," *Proc. ACM SIGCOMM,* pp. 97-108, 2005.

[6] P. Ganesan, B. Yang, and H. Garcia-Molina, "One Torus to Rule Them All: Multidimensional Queries in P2P Systems," *Proc. Seventh Int'l Workshop the Web and Databases (WebDB '04),* pp. 19-24, 2004.

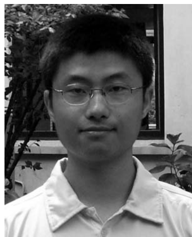
[7] J. Gao and P. Steenkiste, "An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems," *Proc. 12th IEEE Int'l Conf. Network Protocols (ICNP '04),* pp. 239-250, 2004.

[8] H.V. Jagadish, B.C. Ooi, and Q.H. Vu, "BATON: A Balanced Tree Structure for Peer-to-Peer Networks," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* pp. 661-672, 2005.

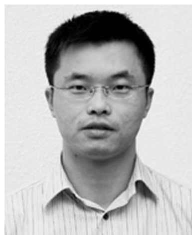
[9] H.V. Jagadish, B.C. Ooi, Q.H. Vu, R. Zhang, and A. Zhou, "Vbi-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes," *Proc. 22nd Int'l Conf. Data Eng. (ICDE '06),* 2006.

[10] 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," *Proc. ACM Symp. Theory of Computing (STOC '97),* pp. 654-663, 1997.

[11] M. Li, W.-C. Lee, and A. Sivasubramaniam, "DPTree: A Balanced Tree Based Indexing Framework for Peer-to-Peer Systems," *Proc. IEEE Int'l Conf. Network Protocols (ICNP '06)*, pp. 12-21, 2006.
[12] S. Ramabhadran, S. Ratnasamy, J.M. Hellerstein, and S. Shenker, "Brief Announcement: Prefix Hash Tree," *Proc. 23rd Ann. ACM Symp. Principles of Distributed Computing (PODC '04)*, 2004.
[13] S. Ratnasamy, P. Francis, M. Handley, R.M. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM SIGCOMM*, 2001.
[14] S.C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling Churn in a DHT," *Proc. USENIX Ann. Technical Conf.*, pp. 127-140, 2004.
[15] 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," *Proc. ACM SIGCOMM*, pp. 73-84, 2005.
[16] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD*, pp. 71-79, 1995.
[17] A.I.T. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *Proc. 18th IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware '01)*, pp. 329-350, 2001.
[18] C. Schmidt and M. Parashar, "Flexible Information Discovery in Decentralized Distributed Systems," *Proc. 12th IEEE Int'l Symp. High Performance Distributed Computing (HPDC '03)*, pp. 226-235, 2003.
[19] G. Shen, C. Zheng, W. Pu, and S. Li, "Distributed Segment Tree: A Unified Architecture to Support Range Query and Cover Query," technical report, Microsoft Research Asia, 2007.
[20] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGCOMM*, pp. 149-160, 2001.
[21] Y. Tang and S. Zhou, "LHT: A Low-Maintenance Indexing Scheme over DHTs," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS '08)*, pp. 141-151, 2008.
[22] Y. Tang, S. Zhou, and J. Xu, "Light: A Query-Efficient Yet Low-Maintenance Indexing Scheme over Dhts," *IEEE Trans. Knowledge Data Eng.*, vol. 22, no. 1, pp. 59-75, Jan. 2010.
[23] P. Yalagandu and J. Browne, "Solving Range Queries in a Distributed System," Technical Report TR-04-18, UT CS, 2003.
[24] C. Zhang, A. Krishnamurthy, and R.Y. Wang, "Brushwood: Distributed Trees in Peer-to-Peer Systems," *Proc. Fourth Int'l Workshop Peer-to-Peer Systems (IPTPS '05)*, pp. 47-57, 2005.
[25] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed Segment Tree: Support of Range Query and Cover Query over DHT," *Proc. Fifth Int'l Workshop Peer-to-Peer Systems (IPTPS '06)*, Feb. 2006.



Yuzhe Tang received the BSc degree, and MSc degree in computer science and engineering from Fudan University, Shanghai, China, in 2006 and 2009. Currently, he is working toward the PhD degree in the College of Computing, Georgia Institute of Technology. His research interests include distributed systems and databases, system security, and privacy.



Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, and the PhD degree in computer science from Hong Kong University of Science and Technology. He is an associate professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include data management for various networked systems including mobile networks, sensor networks, and internet systems. He has published more than 80 technical papers in these areas. He is a senior member of the IEEE.

Shuigeng Zhou received the Bachelor's degree of electronic engineering from Huazhong University of Science and Technology (HUST) in 1988, the Master's degree of electronic engineering from the University of Electronic Science and Technology of China (UESTC) in 1991, and the PhD degree in computer science from Fudan University, in 2000. Now he is a professor in School of Computer Science and Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China. From 1991 to 1997, he served in Shanghai Academy of Spaceflight Technology, as an engineer, and since August 1995, as a senior engineer, respectively. From 2000 to 2002, he was a postdoctoral researcher in State Key Laboratory of Software Engineering, Wuhan University. His research interests include data management in distributed environments, data mining, bioinformatics, and complex networks. He has published more than 100 papers in domestic and international journals and conferences. Currently, he is a member of the IEEE, IEEE Computer Society, the ACM SIGMOD, and the IEICE.

Wang-Chien Lee received the BS degree from the National Chiao Tung University, Hsinchu, Taiwan, the MS degree from the Indiana University, Bloomington, and the PhD degree from the Ohio State University, Columbus. He is an associate professor of computer science and engineering at the Pennsylvania State University, University Park, where he leads the Pervasive Data Access (PDA) Research Group to perform cross-area research in database systems, pervasive/mobile computing, and networking. Prior to joining Pennsylvania State University, he was a principal member of the technical staff at Verizon/GTE Laboratories. His research interests include developing data management techniques (including accessing, indexing, caching, aggregation, dissemination, and query processing) for supporting complex queries in a wide spectrum of networking and mobile environments, such as peer-to-peer networks, mobile ad hoc networks, wireless sensor networks, and wireless broadcast systems. Meanwhile, he has worked on XML, security, information integration/retrieval, and object-oriented databases. His research has been supported by the US National Science Foundation (NSF) and industry grants. Most of his research results have been published in prestigious journals and conference proceedings in the fields of databases, mobile computing, and networking. He has served as a guest editor for several journal special issues on mobile database-related topics, including the *IEEE Transactions on Computers*, *IEEE Personal Communications Magazine*, *ACM MONET*, and *ACM WINET*. He was the founding program committee cochair for the International Conference on Mobile Data Management. He is a member of the IEEE and the ACM.

Dingxiong Deng received the Bachelor's degree in computer science and engineering from East China University of Science and Technology, in 2008. He is currently working toward the Master's degree in the School of Computer Science, Fudan University. His research interests include peer-to-peer networks and spatial database.

Yue Wang is currently an undergraduate student major in computer science and engineering from Fudan University, Shanghai, China. His research interests include peer-to-peer networks, information retrieval, and search engine.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.