LIGHT: A Query-Efficient Yet Low-Maintenance Indexing Scheme over DHTs

Yuzhe Tang
Shuigeng Zhou
Jianliang Xu
Hong Kong Baptist University, xujl@comp.hkbu.edu.hk

This document is the authors' final version of the published article.
Link to published article: http://dx.doi.org/10.1109/TKDE.2009.47

APA Citation
LIGHT: A Query-Efficient yet Low-Maintenance Indexing Scheme over DHTs

Yuzhe Tang, Shuigeng Zhou† Member, IEEE, and Jianliang Xu, Senior Member, IEEE,

Abstract—DHT is a widely used building block for scalable P2P systems. However, as uniform hashing employed in DHTs destroys data locality, it is not a trivial task to support complex queries (e.g., range queries and k-nearest-neighbor queries) in DHT-based P2P systems. In order to support efficient processing of such complex queries, a popular solution is to build indexes on top of the DHT. Unfortunately, existing over-DHT indexing schemes suffer from either query inefficiency or high maintenance cost. In this paper, we propose LIGHTweight Hash Tree (LIGHT)—a query-efficient yet low-maintenance indexing scheme. LIGHT employs a novel naming mechanism and a tree summarization strategy for graceful distribution of its index structure. We show through analysis that it can support various complex queries with near-optimal performance. Extensive experimental results also demonstrate that, compared with state-of-the-art over-DHT indexing schemes, LIGHT saves 50%-75% of index maintenance cost and substantially improves query performance in terms of both response time and bandwidth consumption. In addition, LIGHT is designed over generic DHTs and hence can be easily implemented and deployed in any DHT-based P2P system.

Index Terms—Distributed hash tables, indexing, complex queries

1 INTRODUCTION

Distributed Hash Table (DHT) is a widely used building block for scalable Peer-to-Peer (P2P) systems. It provides a simple lookup service: given a key, one can efficiently locate the peer node storing the key. The past few years have seen a number of DHT proposals, such as Chord [1], CAN [2], Pastry [3], and Tapestry [4]. By employing consistent hashing [5] and carefully designed overlays, these DHTs exhibit several advantages that fit in a P2P context:

- 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."
- Robustness: DHTs are resilient to network dynamics and node failures that are common in large-scale P2P networks.
- Load balancing: Load balance in DHTs can be efficiently achieved thanks to uniform hashing.

As a result, several DHT services have been deployed in real life, such as the OpenDHT project [6] and the Kademia DHT [7] for trackerless BitTorrent [8]. While DHTs are popular in developing various P2P applications, such as large-scale data storage [9], [10], [11], content distribution [12], and scalable multicast/anycast services [13], [14], they are extremely poor in supporting complex queries such as range queries and k-nearest-neighbor (k-NN) queries. This is primarily because data locality, which is crucial to processing such complex queries, is destroyed by uniform hashing employed in DHTs.

In this paper, we address a challenging problem of how to efficiently support complex query processing in existing DHT-based P2P systems. This problem is interesting to many real-life P2P applications/services. For example, some third-party developers may want to offer complex query facilities over the public OpenDHT service [6]. For another example, some P2P users may want the Kademia DHT deployed in BitTorrent networks to support such range queries as “finding all trackers with torrents updated in the last three days.”

To tackle the problem, an effective yet simple solution is to build indexes on top of existing DHTs (known as over-DHT indexing paradigm [15]). Several indexing schemes following this paradigm have recently been proposed, including Prefix Hash Trie (PHT) [15], [16], Range Search Tree (RST) [17], and Distributed Segment Tree (DST) [18]. Compared to another category of indexing schemes that entail development of new locality-preserved overlays (known as overlay-dependent indexing paradigm), over-DHT indexing schemes are more appealing to our problem for several reasons. First, over-DHT indexing schemes do not need to modify existing DHT infrastructures, whereas overlay-dependent indexing schemes would need to either change inner structures of existing DHTs or build extra overlays from scratch, both of which significantly increase the complexity of deployment. Second, following the design principle of layering [16], over-DHT indexing schemes are of great simplicity to design and implement; index developers can focus on the design of index structures only, while overlay-dependent indexing schemes rely only on the “put/get/lookup” interfaces of generic DHTs, they are applicable to any DHT-based P2P system. This is particularly appreciated given the fact that today’s DHTs in use or in research differ significantly in their overlay structures [1], [3], [4]. In addition, to support multiple P2P applications, the over-DHT indexing paradigm allows for building multiple indexes on a single DHT infrastructure, thereby minimizing the overall system maintenance cost.

Two issues are critical to the performance of an over-DHT indexing scheme: query efficiency and index maintenance cost. In conventional applications where queries are more frequent than data updates, achieving query efficiency is considered the first priority. However, in P2P systems, peer joins and departures usually result in data insertions and deletions to/from the system; and the peer join/departure rate can be as high as the query rate [19], [20]. Such data updates incur constant index updates. Thus, the cost of index maintenance becomes a non-negligible factor in evaluating system performance. This perspective, however, is not realized in existing over-DHT indexing schemes. On the contrary, existing schemes improve query efficiency by sacrificing...
index maintenance cost as a trade-off. More specifically, in a distributed context, each peer maintains a local view of the global index; in order to achieve a better query performance, the common idea is to enlarge the local view and let each peer know more about the global index. For example, in PHT [15], each leaf node has direct access to its siblings. In DST [18], the index structure remains static and is made known globally. However, this static design inherently goes against the dynamic nature of P2P systems and easily leads to load imbalance. As an alternative, RST [17] allows for dynamic tree growth/contraction and further employs a broadcasting mechanism to maintain its global view. However, the index maintenance cost is prohibitively high as a single node split causes a broadcast to all other nodes, which may render the whole P2P system unscaleable.

In this paper, we propose LIGhtweight Hash Tree (LIGHT) — a low-maintenance yet query-efficient scheme for data indexing over DHTs. Two novel techniques contribute to the superior performance of LIGHT: a clever naming mechanism that gracefully distributes the index structure over the underlying DHT, and a tree summarization strategy that offers each peer a scalable local view without incurring extra maintenance cost. LIGHT can efficiently support various complex queries, including range queries, min/max queries, and k-NN queries. As an over-DHT index, LIGHT requires no modification of the underlying DHT and hence possesses the virtues of simplicity and adaptability.

The contributions made in this paper can be summarized as follows:

- We propose LIGHT to address both query efficiency and maintenance efficiency for data indexing over DHTs.
- We develop efficient algorithms to process range queries, min/max queries, and k-NN queries based on the LIGHT index. We show through analysis that most of these queries can be supported with near-optimal performance (i.e., at most three more DHT-lookups than the optimum).
- We present two enhancements to LIGHT: an extensible technique for indexing unbounded data domains and a double-naming strategy for improving system load balance. To the best of our knowledge, LIGHT is the first over-DHT indexing scheme with such flexibility.
- We conduct extensive experiments to evaluate the performance of LIGHT. Compared with state-of-the-art indexing schemes, namely PHT [15], [16] and DST [18], LIGHT saves 50%-75% of maintenance cost and substantially improves query performance in terms of both response time and bandwidth consumption.

The rest of this paper proceeds as follows. Section 2 surveys related work. Section 3 presents the LIGHT index structure, followed by a description of its lookup operation in Section 4. How to update the LIGHT index is explained in Section 5. Section 6 gives the algorithms for processing various complex queries based on the LIGHT index. Section 7 experimentally evaluates the performance of LIGHT. Enhancements to LIGHT are discussed in Section 8. Finally, Section 10 concludes this paper.

2 RELATED WORK

P2P data indexing has recently attracted a great deal of research attention. Existing schemes can be classified into two categories: over-DHT indexing paradigm and overlay-dependent indexing paradigm. While over-DHT indexing schemes treat data indexing as an independent problem free from the underlying P2P substrates, overlay-dependent indexing schemes are intended to closely couple indexes with the overlay substrates.

In this section, we start with a brief overview of DHT overlays, followed by a detailed survey of existing P2P indexing schemes. Here, only structured P2P networks are considered.

2.1 Scalable DHT Overlays

In the design of DHT overlays, the primary concern is topological scalability in terms of two aspects: the diameter, which determines the bound of the hops of a lookup operation, and the degree, which determines the size of the routing table. Many proposed DHT overlays, including Chord [1], Pastry [3], Tapestry [4], and Bamboo [21], are based on the Plaxton Mesh [22], which achieves a diameter of \((\beta - 1) \log_d N\) and a degree of \(\log_d N\). Here, \(\beta\) indicates the base of the DHT identifier space, for example, \(\beta = 2\) in Chord. Another classical DHT, CAN [2], leverages the d-torus topology, which bears a diameter of \(\frac{1}{2} d N^\frac{1}{d}\) and a degree of \(2d\). From a graph-theory viewpoint, given diameter \(k\) and degree \(d\), the number of nodes in a graph, \(N\), is bounded by the Moore bound [23], that is, \(N \leq 1 + d + d^2 + \cdots + d^k\). However, the Moore bound is not generally achievable. To move towards this optimal bound, several DHT overlays were inspired from the topologies of de Bruijn graphs [24], [25], butterfly graphs [26], and Kautz graphs [27]. A more thorough analysis of DHTs regarding scalability and fault tolerance can be found in [25].

2.2 Over-DHT Indexing Paradigm

In the over-DHT indexing paradigm, the DHT and data are loosely coupled by the keys (called DHT keys) generated from data records. Thus, a critical issue in the design of an over-DHT index is how to generate the DHT keys regarding data locality. In this category, the Prefix Hash Trie (PHT) [15], [16] is a representative solution for range queries, and is the most relevant scheme to our proposed LIGHT. Thus, below we first introduce PHT in detail. After that, we present other indexing schemes that support various queries for database and information retrieval applications.

PHT: As the first over-DHT index proposal, PHT supports indexing bounded one-dimensional data. Essentially, PHT partitions the indexing space with a trie (prefix tree) structure, where all data records are stored on leaf nodes. The trie structure is materialized over the DHT in a straightforward way — all tree nodes, including internal nodes and leaf nodes, are mapped into the DHT by directly hashing their labels of binary representation. A splitting/merging process is triggered for PHT whenever overload/underload occurs on leaf nodes. During the splitting process, two children with new labels are generated and hence all data records in the parent node need to be re-located according to the new labels. On the contrary, all data records in the children nodes would be re-located according to the parent’s label during a merging process. The range query processing in PHT involves forwarding the query from the root to all candidate leaf nodes in parallel. To facilitate traversing candidate leaf nodes, PHT further maintains links between neighboring leaves, which however incur extra index maintenance overhead. Due to its simplicity and adaptability, PHT has been deployed in real-world applications [16]. RandPeer [28] applied PHT to a specific scenario — indexing membership data for QoS-sensitive P2P applications.

Other Indexing Schemes for Range/k-NN Queries: Several other studies have also investigated data indexing for range and k-NN queries, with their major focus being how to improve query latency by data replication. DST [18] replicates data records across all ancestors of a leaf node in the trie structure. To process a range query, DST decomposes the range into several subranges, each corresponding to an internal node. Since the trie structure is static and globally known, the internal nodes can be located by a single DHT-lookup, rendering
the range query solved in $O(1)$ time. However, due to data replication in all ancestors, some high-level tree nodes could easily be overloaded. To address this issue, RST [17] employs a novel data structure, called LBM (Load Balancing Matrix), which organizes overloaded tree nodes into a matrix by further replication/partition. The nodes in LBM are mapped into the underlying DHT by hashing the internal labels as well as the matrix coordinates. As for query processing, due to query skewness, range queries are usually distributed only on a portion of the internal nodes (called query band). Based on this observation, a dynamic RST is proposed to adapt the tree structure to the current query band, making the index more efficient and the query load more balanced. To maintain a global view of this dynamic index, however, it relies on a broadcasting mechanism, which is bandwidth-consuming and unscalable in terms of index maintenance cost.

To support multi-dimensional query processing, a naive solution is to employ multiple indexes, one for each dimension, as in RST. However, this solution not only increases index maintenance overhead but also complicates query processing. A later version of PHT [16] leverages space-filling curves to reduce dimensions. PRISM [29] employs reference vectors to generate DHT keys for multi-dimensional data. Following the tree maintenance method of RST, DKDT [30] embeds the k-d tree to support 2-d similarity search. Chen et al. [31] suggested a framework for range indexing and proposed various strategies for mapping tree-based index structures into DHTs. Tanin et al. [32] superimposed the quadtree over the DHT for spatial indexing and querying. Each quadtree node is mapped into the DHT by hashing its centroid. While this paper focuses on one-dimensional data indexing, our proposed LIGHT scheme can nevertheless be extended to multi-dimensional data indexing by employing, for instance, dimension reduction techniques through space-filling curves [16].

Join and Keyword Queries: Join queries have attracted considerable research attention in P2P database systems [33], [34], [35]. While focusing on different types of equi-joins (e.g., two-way vs. multi-way, snapshot vs. continuous joins), they generally allocate data records by hashing both the names and values of join attributes, and aim to map the joining records (the records with the same value on the join attributes) to the same DHT node. For these P2P database systems, LIGHT can be seamlessly integrated by indexing the join columns to support general range-based joins, since the essence of such joins consists of range queries in a 2-level nested loop.

While databases cope with structured data, there are other systems that deal with semi-structured or unstructured data such as XML, RDF, and text. In these systems, processing effective keyword queries is essential. To support them over DHTs, a typical solution is to employ the Distributed Inverted Index (DII) [36]. In DII, the inverted index is superimposed over the DHT by directly hashing indexed keywords, and posting lists are intersected for conjunctive keyword search. The major problems of DII are that, due to the Zipf distribution of text keywords, direct keyword hashing results in load imbalance; and due to destroyed data locality (particularly the keyword correlation) by hashing, intersecting posting lists consumes lots of bandwidth. To address these problems, many techniques have been proposed [36], [37], [38]. Following the framework of DII, Cai and Frank [39] and Galanis et al. [40] have also studied RDF and XML data indexing over DHTs.

2.3 Overlay-dependent Indexing Paradigm

In the overlay-dependent indexing paradigm, the overlay substrate directly bears data locality. The existing schemes generally follow two approaches: LSH (Locality Sensitive Hashing) and indexable overlays.

LSH-based Indexing: Rather than using uniform hashing, LSH-based indexing employs locality-sensitive hashing to map data into the overlay in a locality preserved way. By this means, some DHT overlays can directly support efficient range query processing [41], [42], [43], [44], [45]. Gupta et al. [46] applied LSH to DHT-based range indexing and provided approximate range query answers. For efficient similarity queries in P2P systems, LSH-Forest [47] refined the traditional LSH by eliminating its data dependence. For keyword search, Joung et al. [48] proposed a novel indexing scheme, in which uniform hashing is replaced with Bloom filtering, and the underlying overlay is modeled as a multi-dimensional hypercube. To conduct keyword search, the hypercube is partially traversed by following a spanning binomial tree. Based on this framework, KISS [49] was developed to support prefix search. While preserving data locality, LSH-based indexing corrupts the uniform key distribution, which leads to load imbalance [50].

Indexable Overlays: Indexable overlays make no use of full-fledged DHTs, but instead re-design the overlay from scratch and map data into it directly. The existing schemes in this category are based on various data structures. Skip graph [51] answers range queries based on a distributed structure originated from skip lists. PTree [52] and PRing [53] are based on distributed B-trees. BATON [54] is an overlay organized as a balanced binary tree. These overlays support one-dimensional data indexing. VBT-Tree [55] is a general framework that aims to map any existing index tree into BATON. It can index multi-dimensional data and support multi-dimensional range queries and k-NN queries. As a similar solution, SD-Rtree [56] uses a distributed balanced binary tree for spatial indexing. Mercury [57] uses a hierarchical ring structure to index multi-dimensional data. In these non-hash schemes, data locality is well preserved at the cost of deteriorated load balance. In recent years, many sophisticated balancing strategies have been proposed [58], [53], [59]. The basic idea is to first locate a light-loaded peer when overload occurs, and then to transfer load between them. These explicit balancing strategies cost much more in maintenance than the DHT hashing methods. In order to correctly locate a light-loaded peer, they typically require an extra overlay, say another skip graph, to index peer load information, which could double the overall overlay maintenance cost. In addition, the load transfer could also be consuming. By contrast, the DHT hashing methods are maintenance-free — once data are allocated by uniform hashing, load balance is statistically guaranteed.

To summarize, the above over-DHT and overlay-dependent indexing schemes all have trade-offs in query/load balancing performance and practical deployment considerations. Although over-DHT indexing schemes are generally less efficient in query performance than overlay-dependent indexing schemes, by following the layering design principle, they are advantageous in terms of simplicity of deployment/implementation/maintenance and inherited load balancing as discussed previously. In practice, these issues are equally important to query performance [16]. Over-DHT indexing schemes are particularly favorable to the applications in which concerns about ease of implementation, deployment and maintenance dominates the need for high query performance. The LIGHT index proposed in this paper follows the over-DHT indexing paradigm and outperforms all existing over-DHT indexing schemes.

3 The LIGHT Index Structure

In this section, we describe the LIGHT index structure and its mapping strategy to the underlying DHT. We remark that LIGHT is proposed to support complex queries over some existing DHTs, while exact-match
queries can be directly and efficiently answered by the existing DHT infrastructure.

3.1 Overview

In the LIGHT index, a data unit is called a record, and each record is identified by a data key (denoted by δ). We assume that the data keys to be indexed fall into a bounded one-dimensional space. Without loss of generality, the space is set to [0, 1] in this paper, and δ can be any floating number in it. On the other hand, to assign the records in the underlying DHT, each data record is associated with another key, called DHT key (denoted by κ). For a given DHT key κ, it is mapped to the peer whose identifier is less than but closest to \(\text{hash}(κ)\). In a naive indexing scheme, one may set the DHT key directly to the data key. However, this would destroy data locality, as mentioned earlier, and lead to inefficient support to complex queries. Thus, similar to other over-DHT indexing schemes, the main challenge of LIGHT is to find a mapping from data keys to DHT keys such that data locality is preserved with minimal maintenance cost. Fig. 1 gives an overview of the mapping operation in LIGHT. First, LIGHT employs a space partition tree to index data keys. Then, after the partition tree is decomposed and summarized in a data structure called a leaf bucket, LIGHT uses a novel naming function to map leaf buckets to DHT keys. In the following, we explain these two procedures in detail.

3.2 Space Partition Tree

As the name implies, the space partition tree (or simply partition tree for short) recursively partitions the data space into two equal-sized subspaces until each subspace contains fewer than \(θ_{\text{split}}\) data keys. Only leaf nodes store data records (or just data entries with pointers pointing to actual data records). Fig. 2 gives an example, where the data frequency histogram is shown at the bottom. We remark that here a data key \(κ\) is critical to complex query processing. Thus, we propose a distributed data structure, termed leaf bucket, to store data records and summarize the partition tree’s structural information.

Fig. 1: LIGHT Indexing Architecture

![Fig. 2: An example of a space partition tree](image)

These two properties collectively guarantee that the number of leaf nodes equals the number of non-leaf nodes. Each node in the tree is assigned a unique label. The virtual root is labeled with a special character “#” in this paper. Each tree edge is labeled with a binary number, 0 (or 1) for the edge connecting the left (or right) child. As a special case, the edge between the virtual root and the ordinary root is labeled with 0. Then for any tree node, its label is the concatenation of the binary numbers on the path from the virtual root to itself (see Fig. 2). To facilitate our further discussions, we define some notations for the partition tree: \(λ\) denotes the label of a leaf node, while \(ω\) denotes the label of an internal node; \(Δ\) denotes the set of the leaves’ labels, that is, \(Δ = \{λ\}\), while \(Ω\) denotes the set of the internal nodes’ labels, that is, \(Ω = \{ω\}\).

3.3 Local Tree Summarization

Recall that data records are stored in leaf nodes; we need to map only leaf nodes to the underlying DHT. On the other hand, a bare leaf node lacks the knowledge of the overall tree structure, which, as we will see, is critical to complex query processing. Thus, we propose a distributed data structure, termed leaf bucket, to store data records and summarize the partition tree’s structural information.

Fig. 2: An example of a space partition tree

Fig. 3: Leaf bucket of the node \#0100

![Fig. 3: Leaf bucket of the node \#0100](image)

Each leaf bucket corresponds to a leaf node in the tree. As illustrated in Fig. 3a, a bucket consists of two fields: leaf label, which maintains the label \(λ\) of the leaf node, and record store, which keeps all data records of the leaf node. For each leaf bucket, the label \(λ\) provides a local view of the partition tree, which is called a local tree. As shown in Fig. 3b, the local tree of leaf node \#0100 consists of all its ancestors and their direct children (termed branch nodes). The label of any node in the local tree can be inferred directly from \(λ\): the label of each ancestor is a prefix of \(λ\), and the label of each branch node can be obtained by inverting the ending bit of a prefix of \(λ\). According to the completeness property of the partition tree, all branch nodes must exist in the tree. Some branch nodes may contain a subtree, called the neighboring subtree, as depicted by shaded triangles in Fig. 3b. The
structures of these neighboring subtrees are unknown in the current local tree, but are maintained by some other leaves’ local trees. From a global viewpoint, the local trees of all leaves together guarantee the partition tree’s integrity. In other words, the leaf buckets collectively maintain the tree’s structural information. Thus, the remaining issue is how to map each leaf bucket as an atomic unit to the DHT key, which is achieved by a novel naming function.

3.4 Naming Function

For a leaf bucket with label $\lambda$, the naming function $f_n(\cdot)$ generates its DHT key, that is, $\kappa = f_n(\lambda)$.

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

$$f_n(\lambda) = \begin{cases} p0 & \text{if } \lambda = \#011^*, \\ p1 & \text{if } \lambda = \#100^*, \\ \# & \text{if } \lambda = \#0^*. \end{cases}$$

where $p = \#0[01]^*$. That is, if $\lambda$ ends up with consecutive 0’s, $f_n(\lambda)$ truncates all the 0’s in the end. Otherwise, it truncates all the 1’s. For example, $f_n(\#01100) = \#011$ and $f_n(\#01011) = \#010$.

In a tree’s view, each $\lambda$ represents a leaf node, and interestingly, each $f_n(\lambda)$ represents a distinct internal node. Fig. 4 illustrates the intuition, in which each leaf bucket $\lambda$ is “named” to an internal node $f_n(\lambda)$ by a dotted arrow, for instance, $f_n(\#011111) = \#0$. This nice property originates from the double-root and completeness properties of the partition tree. Recall that $\Lambda$ and $\Omega$ represent the sets of labels for internal nodes and leaf nodes, respectively. We obtain the following theorem.

**Theorem 3.1:** $f_n(\cdot)$ is a bijective mapping from $\Lambda$ to $\Omega$.

**Proof:** We first prove that $f_n(\cdot)$ is indeed a mapping from $\Lambda$ to $\Omega$, and then prove that $f_n(\cdot)$ is bijective.

For $\forall \lambda \in \Lambda$, $f_n(\lambda)$ is a prefix of $\lambda$. By the labelling strategy, any prefix of $\lambda$ represents an ancestor of the corresponding leaf, which in other words is an internal node. Therefore, $f_n(\cdot)$ is a mapping from $\Lambda$ to $\Omega$.

As for the bijection, we prove a more concrete proposition: for $\forall \omega \in \Omega$, there is one and only one $\lambda$ mapped to it. First consider the special case where $\omega$ is the virtual root; by definition, the leaf mapped to the virtual root is the leftmost leaf (i.e., the leaf labeled as $\#0^*$). For any other internal node $\omega$, there are two cases: $\omega$ ends up with a 0 (i.e., $\omega = \omega'0$), or with a 1 (i.e., $\omega = \omega'1$). For the first case, the leaf that is mapped to $\omega$ must be labeled as $\omega'1*$, because $f_n(\omega'1*) = f_n(\omega'011*) = \omega$ (line 2). Then we prove that $\forall x, \mu$, $f_n(\lambda, (x, \mu))$ is $f_n(x, \mu)$, as defined blow.

**Definition 2:** For $\forall x \in \Gamma(\mu)$, the next naming function $f_n(x, \mu)$ is

$$f_n(x, \mu) = \begin{cases} p0 & \text{if } x = p0, \\ p1 & \text{if } x = p1. \end{cases}$$

where $p = \#0[01]^*$. Intuitively, $f_n$ locates the first bit in the suffix of $\mu$ (with respect to $x$) that differs from $x$’s ending bit; the value $f_n(x, \mu)$ is then the prefix of $\mu$, which ends up with this located bit. For example, $f_n(\#001, \#011110) = \#011110$.

Note that the prefixes between $x$ and $f_n(x, \mu)$ all share the same DHT key, namely $f_n(x)$. In the above example, $f_n(\#001) = f_n(\#00111) = f_n(\#01111) = \#00$. Thus, there is no point in searching the prefixes $\#00011$ and $\#00111$, since $\#001$ has been checked.

**An example:** Consider a lookup of 0.9 with $D = 14$. Suppose LIGHT is as shown in Fig. 2 and the target bucket is the leaf $\#01110$. Note that $\mu(0.9, 14) = \#01110011001110$. Initially, the lower bound is 2 and the upper bound is 15. LIGHT first tries the prefix of half length, that is, $\#0111001$, and performs a DHT-get for

4. In this paper, we may refer to LIGHT lookup as “lookup” for short, and for clarity the DHT-lookup remains its full name.

5. As done in PHT [15] and another range query scheme [60], $D$ can be obtained by estimating the size and distribution of the indexed dataset.
Algorithm 1 LIGHT-lookup(data key $\delta$)

1: $\mu \leftarrow \text{binary-convert}(\delta)$
2: lower $\leftarrow 2$, upper $\leftarrow D + 1$
3: while lower $\leq$ upper do
4: mid $\leftarrow \left(\text{lower}+\text{upper}\right)/2$
5: $x \leftarrow \mu.\text{prefix}(\text{mid})$
6: $\text{bucket-label} \leftarrow \text{DHT-get}(f_{\mu})(x)$
7: if $\text{bucket-label} = \text{NULL}$ then [a failed DHT-get]
8: upper $\leftarrow f_{\mu}(x).\text{length}$
9: else if $\text{bucket-label}$ covers $\delta$ then [reach the target leaf bucket]
10: return $f_{\mu}(x)$
11: else $x$ is an ancestor of the target leaf node)
12: lower $\leftarrow f_{\mu}(x, \mu).\text{length}$
13: return NULL

Algorithm 2 leaf-split(leaf bucket $b$)

1: $\lambda \leftarrow b.\text{leaflabel}$
2: if $\lambda = \#011$ then
3: $r_b.\text{leaflabel} \leftarrow \lambda 0$ \{rb is the remote leaf bucket\}
4: $b.\text{leaflabel} \leftarrow \lambda 1$
5: else
6: $r_b.\text{leaflabel} \leftarrow \lambda 1$
7: $b.\text{leaflabel} \leftarrow \lambda 0$
8: Add corresponding records to $r_b$ and delete them in $b$.
9: Locally write $b$ back to the disk of current peer.
10: DHT-put($\lambda$, $r_b$)

Algorithm 2 formally describes how the leaf bucket splits in a distributed manner. The procedure leaf-split($b$) is invoked whenever a leaf bucket $b$ is found containing $\theta_{\text{split}}$ or more records during a data insertion. In order to split, it first checks the value of the leaf label $\lambda$ (line 2) and accordingly updates the labels of $b$ and the remote leaf bucket $r_b$ (lines 3–7). Since the data space is partitioned, the records are reassigned into $b$ and $r_b$ (line 8). After updating $b$ locally, the algorithm calls a DHT-put to place the bucket $r_b$ onto some remote peer (line 10). During the whole process, the algorithm relies only on local knowledge and consumes one DHT-lookup (in the DHT-put).

5.1 Data Insertion and Leaf Split

Inserting a data record into LIGHT involves a LIGHT lookup and a possible leaf split process. More specifically, for a data key $\delta$, LIGHT performs a lookup to locate the target leaf bucket $\lambda(\delta)$, and then calls a DHT-put to place the record there. However, if the leaf bucket $\lambda(\delta)$ is already full (i.e., containing $\theta_{\text{split}}$ or more records), the insertion will split the bucket and generate two new leaves. In LIGHT, one leaf bucket will stay on the current peer, denoted as the local leaf, while the other one, denoted as the remote leaf, will be pushed out to some other peer. The local leaf is not pushed out and consumes no bandwidth overhead. This nice property, which we call incremental leaf split, is explained in Theorem 5.1.

Theorem 5.1: Consider a leaf labeled with $\lambda$, which would be split into two nodes, labeled with $\lambda 0$ and $\lambda 1$. The naming function maps one still to $f_{\lambda}(\lambda)$, and the other one to $\lambda$.

Proof: We consider the case where $\lambda$ ends with 0 (the case where $\lambda$ ends with 1 can be proved similarly). Without loss of generality, we assume that $\lambda = \#0[01][0]*100\ast$. After the split, the labels of the two new nodes, $\lambda 0$ and $\lambda 1$, are $\#0[01][0]*100\ast$ and $\#0[01][0]*100\ast 1$, respectively. Obviously, $f_{\lambda}(\lambda 0) = f_{\lambda}(\#0[01][0]*100\ast) = f_{\lambda}(\#0[01][0]*100\ast) = f_{\lambda}(\lambda 1)$, and $f_{\lambda}(\lambda 1) = f_{\lambda}(\#0[01][0]*100\ast 1) = \lambda$.

Theorem 5.1 directly leads to incremental leaf split: by LIGHT’s mapping strategy, the previous leaf bucket $\lambda$ is mapped to the peer with identifier hash$_\lambda(f_{\lambda}(\lambda))$. After splitting, the local leaf bucket is still named to $f_{\lambda}(\lambda)$ and thus remains on the same peer. The other one, which is named to $\lambda$, gets a new name and is mapped to some remote peer. Such a process is illustrated in Fig. 6.

5.2 Data Deletion and Leaf Merge

To remove the data key $\delta$, LIGHT, similar to data insertion, first performs a lookup to locate the leaf bucket that covers $\delta$, say $\lambda(\delta)$. It then executes a local deletion operation to remove the corresponding record.

Data deletion may further lead to a merge of leaf buckets if the number of records (called load for brevity) contained in the leaf and joins/​departures/​failures), the LIGHT index only needs to handle data updates while leaving network structure changes to the underlying DHT. In this section, we present the LIGHT index maintenance algorithms for data insertions and deletions.

5 LIGHT MAINTENANCE

Unlike overlay-dependent indexes that would update their structures with network structure changes caused by system dynamics (i.e., peer joins/​departures/​failures), the LIGHT index only needs to handle data updates while leaving network structure changes to the underlying DHT. In this section, we present the LIGHT index maintenance algorithms for data insertions and deletions.

5.1 Data Insertion and Leaf Split

Inserting a data record into LIGHT involves a LIGHT lookup and a possible leaf split process. More specifically, for a data key $\delta$, LIGHT performs a lookup to locate the target leaf bucket $\lambda(\delta)$, and then calls a DHT-put to place the record there. However, if the leaf bucket $\lambda(\delta)$ is already full (i.e., containing $\theta_{\text{split}}$ or more records), the insertion will split the bucket and generate two new leaves. In LIGHT, one leaf bucket will stay on the current peer, denoted as the local leaf, while the other one, denoted as the remote leaf, will be pushed out to some other peer. The local leaf is not pushed out and consumes no bandwidth overhead. This nice property, which we call incremental leaf split, is explained in Theorem 5.1.

Theorem 5.1: Consider a leaf labeled with $\lambda$, which would be split into two nodes, labeled with $\lambda 0$ and $\lambda 1$. The naming function maps one still to $f_{\lambda}(\lambda)$, and the other one to $\lambda$.

Proof: We consider the case where $\lambda$ ends with 0 (the case where $\lambda$ ends with 1 can be proved similarly). Without loss of generality, we assume that $\lambda = \#0[01][0]*100\ast$. After the split, the labels of the two new nodes, $\lambda 0$ and $\lambda 1$, are $\#0[01][0]*100\ast$ and $\#0[01][0]*100\ast 1$, respectively. Obviously, $f_{\lambda}(\lambda 0) = f_{\lambda}(\#0[01][0]*100\ast) = f_{\lambda}(\#0[01][0]*100\ast) = f_{\lambda}(\lambda 1)$, and $f_{\lambda}(\lambda 1) = f_{\lambda}(\#0[01][0]*100\ast 1) = \lambda$.

Theorem 5.1 directly leads to incremental leaf split: by LIGHT’s mapping strategy, the previous leaf bucket $\lambda$ is mapped to the peer with identifier hash$_\lambda(f_{\lambda}(\lambda))$. After splitting, the local leaf bucket is still named to $f_{\lambda}(\lambda)$ and thus remains on the same peer. The other one, which is named to $\lambda$, gets a new name and is mapped to some remote peer. Such a process is illustrated in Fig. 6.

Algorithm 2 leaf-split(leaf bucket $b$)

1: $\lambda \leftarrow b.\text{leaflabel}$
2: if $\lambda = \#011$ then
3: $r_b.\text{leaflabel} \leftarrow \lambda 0$ \{rb is the remote leaf bucket\}
4: $b.\text{leaflabel} \leftarrow \lambda 1$
5: else
6: $r_b.\text{leaflabel} \leftarrow \lambda 1$
7: $b.\text{leaflabel} \leftarrow \lambda 0$
8: Add corresponding records to $r_b$ and delete them in $b$.
9: Locally write $b$ back to the disk of current peer.
10: DHT-put($\lambda$, $r_b$)

Algorithm 2 formally describes how the leaf bucket splits in a distributed manner. The procedure leaf-split($b$) is invoked whenever a leaf bucket $b$ is found containing $\theta_{\text{split}}$ or more records during a data insertion. In order to split, it first checks the value of the leaf label $\lambda$ (line 2) and accordingly updates the labels of $b$ and the remote leaf bucket $r_b$ (lines 3–7). Since the data space is partitioned, the records are reassigned into $b$ and $r_b$ (line 8). After updating $b$ locally, the algorithm calls a DHT-put to place the bucket $r_b$ onto some remote peer (line 10). During the whole process, the algorithm relies only on local knowledge and consumes one DHT-lookup (in the DHT-put).

5.2 Data Deletion and Leaf Merge

To remove the data key $\delta$, LIGHT, similar to data insertion, first performs a lookup to locate the leaf bucket that covers $\delta$, say $\lambda(\delta)$. It then executes a local deletion operation to remove the corresponding record.

Data deletion may further lead to a merge of leaf buckets if the number of records (called load for brevity) contained in the leaf and
its sibling drops below $\theta_{\text{split}}$. Algorithm 3 illustrates the leaf merge operation. In line 1, we predefine a merge threshold $\theta_{\text{merge}}$, which determines whether to trigger a probe of its sibling’s load (line 2). The sibling node is located by a sibling function $f_s$, which is defined as below:

**Definition 3:** For a leaf bucket labeled with $\lambda$, the sibling function $f_s(\lambda)$ returns the DHT key of its sibling.

$$f_s(\lambda) = \begin{cases} f_n(p_0) & \text{if } \lambda = p_1, \\ f_n(p_1) & \text{if } \lambda = p_0. \end{cases}$$

where $p = \#(0|1)^*$. The algorithm proceeds to check whether $\lambda$’s sibling is a leaf node, that is, whether the retrieved label $\lambda_s$ for the sibling has the same length as $\lambda$ (line 3). If this is true and their total load is lower than $\theta_{\text{split}}$, the actual merge operation is started (lines 5–9). Note that in the push operation, there is no need to perform the DHT-lookup for $f_s(\lambda)$ again, since it was already found in line 2. Thus, only one DHT-lookup is incurred for each data deletion operation.

By the definition of our space partition tree, $\theta_{\text{merge}} = \theta_{\text{split}}$. In practice, however, to avoid unnecessary checking and hence save bandwidth consumption, $\theta_{\text{merge}}$ can be set to a fraction of $\theta_{\text{split}}$ (e.g., half of $\theta_{\text{split}}$), though this would deteriorate the index consistency.

### 5.3 Analysis of Tree Maintenance Cost

#### 5.3.1 Cost Model

Before analyzing the tree maintenance cost, we propose a cost model reasonable for over-DHT indexing schemes. A P2P network is characterized by abundant local resources. That is, a typical P2P network holds ample resources (e.g., local disk storage and CPU power) at the network edges. By contrast, the inter-network resource, namely, the bandwidth, is relatively rare and thus critical in a P2P network. Therefore, to capture the P2P network cost, we consider only the bandwidth consumption in the analysis. For an over-DHT indexing scheme, two operations are bandwidth-consuming: DHT-lookup and data movement (i.e., transferring data records from one peer to another via a physical connection, like TCP or UDP). We assume that moving each data record costs $\iota$ units and each DHT-lookup costs $\jmath$ units. The value of $\iota$ is determined by the size of a data record — the larger the data record, the higher the cost incurred for data transferring. The value of $\jmath$ is determined by the scale of the underlying P2P network — for a P2P network with more peers, a DHT-lookup incurs more physical hops (typically at complexity of $O(\log N)$), which leads to a larger $\jmath$.

#### 5.3.2 Maintenance Cost

In the interest of space, only data insertion is discussed here; data deletions can be similarly analyzed. As discussed earlier, each data insertion involves a LIGHT lookup and a possible leaf split. A LIGHT lookup incurs $\log(D/2)$ DHT-lookups and movement of one data record.

For each leaf split in LIGHT, only one DHT-lookup is incurred, yielding the DHT-lookup cost of $\jmath$; the data-movement cost is proportional to the size of the remote leaf bucket. Note that for a pair of remote and local buckets, their sizes sum to $\theta_{\text{split}}$. Let the size of the remote bucket be a fraction of $\theta_{\text{split}}$, denoted as $\alpha \cdot \theta_{\text{split}}$, where $\alpha$ is a normalized factor in $[0, 1]$. The size of the local bucket is thus $(1-\alpha) \cdot \theta_{\text{split}}$. For a specific split, the very value of $\alpha$ is determined by the local data distribution on the splitting node. For a large enough tree, while the data skewness does affect the global tree structure, the local data distribution within a leaf node is likely to be uniform, yielding an average $\alpha$ equal to $\frac{1}{2}$. Thus, the average data-movement cost per split is $\frac{1}{2} \theta_{\text{split}} \cdot \iota$. In all, the average cost for one leaf split in LIGHT, denoted as $\Psi_{\text{LIGHT}}$, is

$$\Psi_{\text{LIGHT}} = \frac{1}{2} \theta_{\text{split}} \cdot \iota + 1 \cdot \jmath.$$ 

We compare the maintenance cost for a leaf split with PHT [15], which is the state of the art with respect to maintenance efficiency. In PHT, an index tree similar to the space partition tree is maintained and, as mentioned, its mapping to the 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 leaf buckets with new labels, which are both mapped to some remote peers. This incurs two DHT-lookups and movement of $\theta_{\text{split}}$ data records. Additionally, a split incurs two extra DHT-lookups to update its B+ tree leaf links [15]. Altogether, the bandwidth cost for a PHT split is

$$\Psi_{\text{PHT}} = \theta_{\text{split}} \cdot \iota + 4 \cdot \jmath.$$ 

In comparison with PHT, LIGHT saves the maintenance cost for leaf split by

$$1 - \frac{\Psi_{\text{LIGHT}}}{\Psi_{\text{PHT}}} = \frac{3}{2} \gamma + 3 \gamma + 4$$

where $\gamma = \frac{\theta_{\text{split}}}{\alpha}$. This savings ratio can range from 50% to 75%, depending on the value of $\gamma$.

### 6 Complex Queries

In this section, we discuss the processing of various complex queries over the LIGHT index, including range queries, min/max queries, and k-NN queries.

#### 6.1 Range Queries

Given two bounds, $l$ and $u$, a range query returns all data records whose keys fall in the range of $[l, u)$. Thanks to the local tree, LIGHT can support range query processing at near-optimal cost. To illustrate how it works, we start with a simple case.
In the general case, the query issuer can be any leaf bucket. As described in Algorithm 5, after receiving the range query $R = [l, u)$, the leaf locally computes the lowest common ancestor that covers $R$, abbreviated as LCA. It then forwards the query by a DHT-lookup of $f_n(LCA)$. We discuss three possible cases: 1) The DHT-lookup has failed (line 3), implying that range $R$ is so small that a single leaf completely covers it. In this case, the range processing is reduced to a lookup operation. 2) The returned leaf bucket overlaps the query range (line 6), implying one range bound must be in this leaf bucket. This is the simple case we discussed above. 3) The returned leaf bucket does not overlap the query range (line 8). In this case, the query range is subdivided and respectively forwarded to LCA's children, namely, LC0 and LCA0. Note that each of the leaves named to LC0 and LCA0 must cover one bound of the corresponding subrange. Thus, the processing of both subsequent queries can follow the simple-case strategy.

**An example:** Consider the range query $[0.3, 0.6)$ on the tree shown in Fig. 7b. Any leaf bucket receiving the query locally calculates the LCA to be #0 and performs a DHT-lookup of $f_n(#0) = #$. The returned leaf bucket is #000, whose range does not overlap the queried range (i.e., the Case 3). As mentioned, the queried range is then subdivided and forwarded to DHT keys #00 (= $f_n(#000) = f_n((#000))$) and #01 (= $f_n(#001)$), to which leaf buckets #0011 and #0100 are respectively named. Bucket #0011 has its
bound value of 0.5 in the queried range and hence the recursive forwarding process then applies; bucket # 0011 further forwards it to # 001 (= fn(fRR(# 0011))), which is the name of bucket # 0010. After that, all leaf buckets in the range [0.3, 0.6] are found.

6.1.3 Complexity
Suppose the query range is distributed on B leaf buckets. We here consider only the case where B ≥ 2 (i.e., Cases 2 and 3 discussed in the last subsection). In general forwarding (Case 3), there is at most one DHT-lookup that returns a leaf bucket not overlapping the range. Moreover, as explained in Section 6.1.1, in the procedure of each recursive forwarding, there is at most one failed DHT-lookup. Therefore, a total of three extra DHT-lookups can possibly occur, that is, the LIGHT-based range query costs at most B + 3 DHT-lookups, which is close to the optimal performance (i.e., B DHT-lookups).

6.2 Range Queries with Lookahead
To further reduce the query latency, we propose a parallel processing algorithm. The basic idea is that each recursive forwarding in the range query looks one step ahead. That is, for each branch node β_i (i = 1, 2, ⋅ ⋅ ⋅ , k) in Fig. 7a, the bucket λ(i) forwards the query not only to fn(β_i) but also to β_i. By this means, each recursive forwarding can explore the neighboring subtree by two levels (instead of one level as in the original algorithm). Therefore, total latency can be reduced by a factor of two. However, the lookahead can increase the number of DHT-lookup failures, typically from 3 to B/2. This is because in the worst case the leaf bucket range may result in a DHT-lookup failure. As such, the lookahead strategy trades bandwidth overhead for shorter query latency. In general, if we look h steps ahead, the overall latency can be reduced by a factor of h + 1, while the number of DHT-lookups is increased by h times. In practice, the user can tune the parameter of h based on his/her performance preferences.

6.3 Min/Max Queries
The min (max) query returns the smallest (largest) data key in the data set. Interestingly, LIGHT supports processing a min/max query at constant cost, owing to its nice naming function. More specifically, the query complexity is one DHT-lookup only.

**Theorem 6.1:** In LIGHT, a DHT-lookup of # returns the smallest key, whereas a DHT-lookup of #0 returns the largest key.

**Proof:** The leaf bucket containing the smallest data key in LIGHT should be the one labeled #0. By the naming function, this bucket #0 is mapped to #. Likewise, the largest data key should be associated with leaf bucket #01, which is named to #0.

6.4 k-NN Queries
Given a data key δ and an integer k, the k-NN query returns the k nearest data keys to δ. LIGHT supports k-NN query processing by a LIGHT lookup of δ, followed by a sequential leaf traversal. Specifically, after the bucket covering δ is located, a bidirectional leaf traversal is set off simultaneously towards the left and the right.

Without loss of generality, we focus on the traversal towards the right. The packet in the leaf traversal carries a parameter unf, which is an integer indicating how many keys still need to be found. It is initiated to 0 and at any time, unf ≤ k. Suppose bucket b receives a k-NN query message of unf and data key δ. As described in Algorithm 6, it locally searches the nearest unf data keys to δ (line 1). Bucket b then returns the results directly to the query issuer (via a physical hop since the query issuer’s address can be known from the packet header) (line 2). The query issuer will update the value of unf according to the current result set and notify bucket b of the new unf' (line 3). If the new unf' is still bigger than 0, meaning that the current result set is not yet filled up, bucket b further forwards the query to its immediate right neighbor (lines 4–10). This is quite similar to the forwarding to β_i in the range query. A k-NN query traversing B buckets incurs at most 1.5B DHT-lookups since in the worst case 50% of DHT-lookups might fail (e.g., the hop from #0011 to #0100 always succeeds but the one from #0100 to #0111 can fail).

**Algorithm 6** k-NN-forward(leaf bucket b, unf, δ)

1. result ← b.localsearch(0, unf)
2. return result to query initiator
3. unf' ← update(unf) {update unf from query initiator}
4. if unf' > 0 then
5. λ ← b.leaflabel
6. β ← fn(λ)
7. nextbucket ← DHT-lookup(β)
8. if nextbucket = NULL then {a failed DHT-lookup}
9. nextbucket ← DHT-lookup(fn(β))
10. k-NN-forward(nextbucket, unf', δ)

7 EXPERIMENTAL RESULTS
This section presents the results of performance evaluation. We compare LIGHT with the state-of-the-art indexing schemes PHT [15] and DST [18] in terms of index maintenance costs and lookup/query performance.

7.1 Experiment Setup

![Fig. 8: Key distribution in DBLP and gaussian dataset](image-url)

We implemented LIGHT in Java. The total number of code lines is 2200 (including LIGHT, DST, and PHT), which demonstrates the simplicity of developing an over-DHT indexing scheme. In the experiments, LIGHT, DST, and PHT were run over the Bamboo DHT [21], a ring-like DHT that has good robustness and is now widely deployed in the OpenDHT project [6]. Our whole system was deployed in a LAN environment consisting of more than 20 computers (or peers).8

Both real data and synthetic data were tested. For the real data, we used the DBLP dataset, which contains the publications listed in the DBLP Computer Science Bibliography.9 The author names were converted to a floating number in the domain of [0, 1] and used as the data keys. By filtering out duplicate author names, we obtained a DBLP dataset containing approximately 250,000 distinct data keys (see Fig. 8 for the data distribution). We further divided the whole dataset into five smaller datasets with 50,000 data keys each. The

8. The performance measurements such as the number of DHT-lookups are independent of the network scale.
9. [URL](http://dblp.uni-trier.de/xml/)
experiments were conducted against all the five small datasets; the average performance is reported here. To evaluate the scalability of the indexing schemes, we also used two synthetic datasets: uniform and gaussian, with sizes varying from 500,000 to 8,000,000. The data keys in the uniform dataset were randomly generated in $[0,1]$, while the data keys in the gaussian dataset follow a gaussian distribution with a mean of $\frac{1}{4}$ and a standard deviation of $\frac{1}{4}$, which guarantees that about 97% of the keys will fall in $[0,1]$ (see Fig. 8). For performance testing on the synthetic data, we repeated each experiment over 30 times and report the average results.

7.2 Structural Properties
In this experiment, we examine the structural properties of the LIGHT index, including average leaf depth, number of leaf nodes, and bucket utilization. Bucket utilization is defined to be the ratio of the number of records stored in a leaf bucket to the bucket capacity $\theta_{split}$. We measure these properties after we inserted 50,000 data keys into the LIGHT index. Fig. 9 shows the performance trends when $\theta_{split}$ is varied from 50 to 1,000. When $\theta_{split}$ grows large, both the average leaf depth and the number of leaf nodes decrease since a large $\theta$ results in leaves containing more keys and thus fewer leaf nodes. Comparing the three datasets under testing, DBLP has more and deeper leaf nodes. This is because the data distribution in DBLP is highly skewed, which makes the index tree very unbalanced. As shown in Fig. 10, most leaf nodes for the uniform dataset have a depth of 13 or 14, whereas the depth of the leaf nodes for DBLP varies from 10 to 25. Fig. 9c shows the bucket utilization as a function of $\theta_{split}$. As expected, the bucket utilization for the DBLP dataset is lowest due to the skewness of data distribution. The bucket utilization for the synthetic datasets, especially the uniform dataset, fluctuates as $\theta_{split}$ increases, owing to the characteristic of the space partition tree.

7.3 Lookup Performance
This experiment evaluates the efficiency of looking up a key in the index. We compare LIGHT with PHT with varying dataset sizes. Note that the lookup operations in both LIGHT and PHT have a parameter $D$, the maximum leaf depth. To make a fair comparison, $D$ is always set to the actual maximum tree depth for the dataset under testing. The splitting threshold $\theta_{split}$ is fixed at the default value 100. For each experiment, we conduct 1,000 lookups for the keys uniformly distributed in $[0,1]$ and record the average number of DHT-lookups per lookup operation. The results are shown in Fig. 11. In general, as expected, the number of DHT-lookups increases as the dataset grows. For the DBLP and gaussian datasets, LIGHT outperforms PHT by 35% on average. For the uniform dataset, the performance curve of PHT exhibits a zigzag shape (see Fig. 11c). This is because most leaf buckets reside in the deepest two levels of the tree (as seen in Fig. 10). As the dataset size is increased, the numbers of leaf buckets on these two levels are increased in turn, for which the binary search gets a fluctuating lookup performance.

7.4 Index Maintenance Performance
We now evaluate the index maintenance performance under data insertions and deletions. In the following, we first compare LIGHT with PHT for the leaf split cost (note that DST incurs no split cost [18]). We then compare the overall index maintenance performance among LIGHT, PHT, and DST.

7.4.1 Leaf Split Costs
In this set of experiments, we first measure the value of $\alpha$ for LIGHT, that is, the ratio of data records moved to remote peers during a leaf split. To evaluate it, we continuously insert data into the LIGHT index and record the average value of $\alpha$ for the leaf splits. As shown in Fig. 12, the average $\alpha$ remains almost constant under different dataset sizes for the uniform and gaussian datasets. For the DBLP dataset, the average $\alpha$ fluctuates a little bit when the dataset size is smaller than 15,000 but becomes stable as the size of the dataset increases. This is mainly because of the irregular distribution of DBLP data (Fig. 8). Fig. 12b shows the result as a function of $\theta_{split}$. In all cases tested,
the average $\alpha$ fairly approaches the value of 0.5, which is consistent with our previous analysis in Section 5.3.2.

Next we compare LIGHT with PHT for the leaf split performance. We continuously insert data into LIGHT and PHT and record the cumulative split costs. Recall that our leaf split involves data-movement costs and DHT-lookup costs (Section 5.3). We measure them separately in each experiment. Figs. 13a and 13b show the results for LIGHT and PHT indexing 50,000 DBLP data keys, with $\theta_{\text{split}}$ varied from 50 to 1,000. For both schemes, total data-movement costs slightly decrease by typically amplifies the insertion cost by a factor of specifically, each insertion in DST needs to look up all the ancestors see that DST incurs a cost higher than LIGHT and PHT by an order includes the costs incurred by both data insertion and leaf split.

7.4.2 Performance under Data Insertions

In this section, we evaluate performance under data insertions, which includes the costs incurred by both data insertion and leaf split. The same experimental settings are chosen as with the leaf split experiments. The results are shown in Figs. 14a and Figs. 14b. We can see that DST incurs a cost higher than LIGHT and PHT by an order of magnitude. This is because DST employs data replication. More specifically, each insertion in DST needs to look up all the ancestors of the leaf and insert the data into the un saturated ancestors, which typically amplifies the insertion cost by a factor of $D$. Comparing LIGHT and PHT, LIGHT still outperforms PHT by about 40% for data-movement costs and 75% for DHT-lookup costs, which conforms to our previous analysis. To further test the scalability, we conduct experiments on the synthetic datasets with varying dataset sizes. From Figs. 13c and 13d, a similar performance improvement can be observed under different dataset sizes for both the uniform and gaussian datasets.

7.4.3 Performance under Data Deletions

We next study performance under data deletions. The experiments proceed in three phases: the growing phase, in which only data insertion is allowed; the steady phase, in which data insertions and deletions are randomly performed; and the shrinking phase, in which data is deleted from the P2P index until it is contracted into a single root. Recall that the leaf merge operation requires a threshold $\theta_{\text{merge}}$. In the experiments, $\theta_{\text{merge}}$ is set to $0.5 \cdot \theta_{\text{split}}$ and $0.2 \cdot \theta_{\text{split}}$. Figs. 15a and 15b show the data-movement costs and DHT-lookup costs respectively for LIGHT and PHT under the DBLP dataset, where the costs for DST are much higher and thus are not shown in the figures for clarity. In Fig. 15a, the data-movement costs remain relatively stable in the steady phase, implying that the split or merge operations rarely happen during this phase. This is because random insertions and deletions may cancel out each other’s effects. Throughout the whole process, the cost of LIGHT remains half that of PHT, and they are both insensitive to the value of $\theta_{\text{merge}}$. In contrast, the DHT-lookup costs in Fig. 15b are more sensitive to $\theta_{\text{merge}}$ — the smaller the $\theta_{\text{merge}}$, the fewer DHT-lookups. This is because a large $\theta_{\text{merge}}$ leads to more DHT-lookups for probing the sibling’s load. Similar performance results are also observed for the uniform and gaussian datasets in Figs. 15c and 15d, where $\theta_{\text{merge}}$ is fixed at $0.5 \cdot \theta_{\text{split}}$.

7.5 Range Query Performance

Finally we evaluate the query processing performance for range queries. The evaluation is in terms of two aspects: time latency and bandwidth costs. The former is captured by the paralleled steps of DHT-lookups, while the latter is captured by the total number of DHT-lookups. Recall that we proposed two LIGHT range query algorithms, the basic one (in Section 6.1) and the one with lookahead (in Section 6.2). PHT also has two range query algorithms, denoted as PHT(sequential) [15] and PHT(parallel) [16], respectively. Thus, we compare these four range query algorithms together with DST.

Fig. 16 shows the range query performance on the DBLP dataset. In Figs. 16a and 16b, the bandwidth costs generally go linearly with the dataset size and the range span. Among the five algorithms, LIGHT(basic) achieves the lowest bandwidth (though not quite visible in the figure), while PHT(sequential) requires a bandwidth slightly higher than LIGHT(basic). As discussed earlier, their performance nearly approaches the optimum, that is, the number of DHT-lookups equals the number of target leaf buckets. The bandwidth costs of PHT(parallel) and DST are twice that of LIGHT(basic) because they both incur internal node traversal when processing range queries. The bandwidth costs of LIGHT(lookahead) are approximately 50% higher than the optimal bandwidth, which again conforms to our previous complexity analysis. In terms of time latency, as shown in Figs. 16c and 16d, the two LIGHT algorithms substantially outperform the others. Without leveraging parallelism, PHT(sequential) incurs extremely high latency. Although parallelism is employed in PHT(parallel) and DST, they still suffer from data skewness for which the deepest leaf node dominates the whole query process.

For the scalability test on the synthetic uniform and gaussian datasets, a similar result is found in Fig. 17. The only exception here is that LIGHT(basic) incurs a slightly higher latency than DST because the skewness is much lower in the synthetic data and DST suits such unskewed distribution.

In summary, LIGHT(basic) outperforms all others in terms of bandwidth costs and achieves quite good time latency, just behind LIGHT(lookahead). LIGHT(lookahead) trades bandwidth for time latency, which makes its time latency the shortest. PHT(sequential) achieves quite efficient bandwidth costs but incurs extremely high latency. PHT( parallel) and DST both incur the highest bandwidth costs, but their latency is not yet the most efficient.

8 Enhancements

In this section, we further present two extensions to the LIGHT index, including how to index unbounded data domains and how to improve peer load balance.

8.1 Extensible Indexing

The basic LIGHT index deals with a bounded data domain (i.e., in the normalized $[0, 1]$ space), which requires a priori knowledge of the

10. For a queried range $[l, u]$, the range span is $u - l$. 
indexed data. However, in many applications, such knowledge cannot be obtained in advance; and even more the data domain may change over time. For example, if we want to index the publication dates of MP3 files in a P2P file-sharing application, the data domain for publication dates is not fixed and evolves in \((-\infty, \infty)\). In this section, we propose E-LIGHT, an extensible LIGHT that supports data indexing of unbounded data domains.

Fig. 18 shows how the E-LIGHT indexes the domain \((-\infty, \infty)\). We introduce two spine buckets, that is, the buckets whose labels are like #00 or #11. For example, in Fig. 18, the right spine bucket
A spine bucket does not employ the binary partition strategy — when it splits, two subspaces can be $[2, 3)$ and $[3, \infty)$. Subspace $[2, 3)$ is then covered by bucket $\#110$, which can subsequently grow into a conventional LIGHT, like $[1, 2)$ indexed by LIGHT $\chi_1$. Essentially, each spine bucket acts as an extending point, and an E-LIGHT index consists of two spine buckets and a set of conventional LIGHTs.

In E-LIGHT, the naming function $f_n(\cdot)$ still applies, as depicted by the dotted arrows in Fig. 18. The only exception is that both spine buckets are named to the root $\#$, which may double the load of the DHT key $\#$. We thus impose the constraint: a spine bucket splits if it stores more than $\frac{2}{3}n$ records. The algorithms for various complex queries in LIGHT can work in E-LIGHT without much modification. For example, given a data key $\delta \in (-\infty, \infty)$, we can easily figure out which LIGHT $\chi_i$ covers the key $\delta$, and afterwards the LIGHT algorithm can be applied to $\chi_i$.

8.2 Improvement of Peer Load Balance

In general, DHTs offer load balance quite efficiently, yet not that effectively. Specifically, if the imbalance ratio denotes the ratio of the heaviest load to the average load for the peers in the P2P network, DHTs only bound the imbalance ratio at $O(\log N)$ with high probability $[5]$, $[1]$. This result is considerably large for large-scale P2P networks. In this section, we propose a double-naming strategy as an improvement for balancing peer load. The double-naming strategy naturally adapts the “power of two choices” paradigm $[61]$ (PoTC) to LIGHT, which bounds the imbalance ratio at $O(\log \log N)$.

The basic idea behind the double-naming strategy is simple. Previously in LIGHT, each leaf bucket $\lambda$ had one name, $f_n(\lambda)$. Now, the double-naming strategy adds another name, that is, $\lambda$ itself. Thus, for each bucket, there are two distinct names, $\lambda$ and $f_n(\lambda)$ (note that $f_n(\lambda) \neq \lambda$), which implies that one bucket can have two choices of underlying peers to store it. Between these two candidates, it picks the one with lighter load to actually store the bucket $\lambda$. Since the load of the peer may change over time, the bucket periodically checks these two peers, and accordingly adjusts its storage location. Unlike PoTC’s other adaptations that come with double hashes within DHTs $[62]$, the double-naming strategy is completely independent of DHTs and can be seamlessly incorporated into LIGHT.

The adaptability comes with prices — to locate the bucket $\lambda$, now two (rather than one) DHT-lookups are needed. For more efficiency, one possible solution is to trade the adaptability. Specifically, a physical link (at the IP level) is maintained between each pair of two candidate peers which, although it incurs modification of underlying DHTs, accelerates the indirection of the failed DHT-lookup and the periodic reassessment of bucket location. So now it consumes one DHT-lookup and a possible physical hop to locate a leaf bucket; this extra hop is trivial, since a typical DHT-lookup incurs $O(\log N)$ physical hops. By this means, various LIGHT algorithms still apply in the double-naming LIGHT, without loss of efficiency.

9 Acknowledgement

We would like to thank the anonymous reviewers. This work is supported by the National Natural Science Foundation of China under Grant Nos. 60496327, 60573183, 90612007 and 60873070, and Shanghai Leading Academic Discipline Project No. B114. Jianliang Xu’s work was supported by grants from the Research Grants Council, Hong Kong SAR, China (Project Nos. HKBU211307 and HKBU210808).

10 Conclusion

This paper proposed LIGHT, a LIGhtweight Hash Tree, for efficient data indexing over DHTs. LIGHT differs from PHT, a representative over-DHT indexing scheme, in the following aspects:

- Both PHT and LIGHT are based on the idea of space partitioning. While PHT maps its index structure into DHT in a straightforward manner, LIGHT leverages a clever naming function, which significantly lowers the maintenance cost and improves the DHT-lookup performance.
- LIGHT employs local tree summarization to provide each bucket a local view. This local view is essentially helpful for distributed query processing, but unlike PHT’s sequential leaf link, requires no extra maintenance cost.
- In PHT, all leaf nodes and internal nodes are mapped to the DHT space, whereas only leaf nodes are mapped in LIGHT. The processing of range queries in PHT has to go through all internal nodes of the subtree in addition to the leaf nodes in the queried range, which at least doubles the search cost.
- Thanks to the novel naming function, one can easily determine the leftmost/rightmost leaf node under a subtree in $O(1)$ lookup. As such, min/max queries can be efficiently supported in LIGHT.
• LIGHT can be extended to index unbounded data domains and naturally accommodate a double-naming strategy to improve peer load balance. As a comparison, PHT (and other existing over-DHT schemes) only supports data indexing of bounded domains and achieves better load balance by modifying DHTs.

Experimental results show that in comparison with the state-of-the-art indexing techniques PHT and DST, LIGHT saves 50% - 75% of index maintenance cost and supports more efficient lookup operations. Moreover, LIGHT has a much better query performance in terms of both bandwidth consumption and response time. As an over-DHT scheme, LIGHT is adaptable to generic DHTs and can be easily implemented and deployed in any DHT-based P2P system.

References


