m-LIGHT: Indexing Multi-Dimensional Data over DHTs

Yuzhe Tang [†] Jianliang Xu [‡] Shuigeng Zhou [†] Wang-Chien Lee [§]

[†]*Fudan University, Shanghai, China, Email:* {yztang, sgzhou}@fudan.edu.cn
[‡]*Hong Kong Baptist University, Kowloon Tong, Hong Kong, Email:* xujl@comp.hkbu.edu.hk
[§]*Pennsylvania State University, PA, USA, Email:* wlee@cse.psu.edu

Abstract

In this paper, we study the problem of indexing multi-dimensional data in the P2P networks based on distributed hash tables (DHTs). We identify several design issues and propose a novel over-DHT indexing scheme called m-LIGHT. To preserve data locality, m-LIGHT employs a clever naming mechanism that gracefully maps the index tree into the underlying DHT so that it achieves efficient index maintenance and query processing. Moreover, m-LIGHT leverages a new data-aware index splitting strategy to achieve optimal load balance among peer nodes. We conduct an extensive performance evaluation for m-LIGHT. Compared to the state-of-the-art indexing schemes, m-LIGHT substantially saves the index maintenance overhead, achieves a more balanced load distribution, and improves the range query performance in both bandwidth consumption and response latency.

1. Introduction

Distributed Hash Table (DHT) provides a scalable, load balanced, and robust substrate in building large-scale distributed applications. Several DHT overlays, such as Chord [1], CAN [2], and Pastry [3], have been proposed. Whereas simple lookup operations can be efficiently executed over DHTs, they lack support for complex queries such as range queries and similarity queries, which are however popular in many P2P applications (e.g., "finding the songs that are rated above 4 and published during 2007 and 2008"). 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 paradigm which builds an add-on index module over generic DHTs (e.g., PHT [4] and DST [5]); 2) DHT-dependent indexing paradigm which modifies the internal structures of underlying DHTs or develops novel locality-preserved overlays (e.g., Skip graphs [6] and BATON [7]). Although the over-DHT indexing approach is generally less efficient in query performance than the DHT-dependent indexing approach, it excels in many other aspects such as simplicity of deployment/implementation/maintenance and inherited load balancing [4], [8], [9], [5]. These issues could be particularly important in practice, for example, if one wants to deploy P2P applications in the world-wide OpenDHT project [10].

In the paper, we study the problem of how to efficiently support multi-dimensional range queries in existing DHT-based P2P systems and thus advocate the over-DHT indexing paradigm. We propose a novel over-DHT index called m-LIGHT (multi-dimensional Lightweight Hash Tree over a DHT). Particularly, we investigate the following problems: 1) how to map a tree-based index into the underlying DHT to better support distributed query processing; and 2) how to perform index maintenance to better balance the loads of peer nodes. We consider multi-dimensional data and employ a space-partition based *kd-tree* to index data [8], [4], [11]. To distribute the kd-tree over a DHT, we propose a *tree-decomposition* strategy that enlarges the local view on each peer yet requires no extra maintenance overhead. We further propose a novel multi-dimensional naming mechanism to gracefully map the decomposed tree into the DHT. The naming mechanism possesses several nice properties that lead to high efficiency in both index maintenance and query processing. Moreover, to address the load balancing issue in space-partition based indexes, we propose a data-aware index splitting strategy to achieve optimal load balance among peer nodes.

The rest of this paper proceeds as follows. Section 2 surveys related work. Section 3 presents the m-LIGHT index structure. How to update the m-LIGHT index is explained in Section 4, followed by a description of its lookup operation in Section 5. Section 6 presents the algorithms for processing range queries based on the m-LIGHT index. Section 7 experimentally evaluates the

performance of m-LIGHT. Finally, Section 8 concludes this paper.

2. Related Work

In this section, we first survey the existing over-DHT indexing schemes, and then review the multi-dimensional indexing techniques, both in a P2P context.

2.1. Over-DHT Indexing

A variety of over-DHT indexing schemes have recently been proposed to support complex queries in P2P systems. Most of them focus on range queries. Prefix Hash Trie (PHT) [4] is the first over-DHT indexing scheme. To perform an exact-match query (or the lookup operation), PHT starts the search from an internal node of the index tree and thus avoids the single-root bottleneck. Internal nodes in PHT do not hold data and serve as routing nodes only. Thus, processing range queries in PHT always needs to traverse down to leaf nodes. An observation here is that if one can fill internal nodes with data, there is no need to traverse down to leaf nodes, thereby accelerating query processing. Following this observation, Distributed Segment Tree (DST) [5] and Range Search Tree (RST) [9] have been proposed. To fill internal nodes, they both replicate the data records of a leaf node at all its ancestors. To process a range query, they decompose the range into several disjoint subranges, each maintained by an internal node. Since each such internal node can be located by a single DHT-lookup, the query can be efficiently resolved in $O(1)$ time. However, the replication strategy could harm the index maintenance efficiency, that is, the insert/delete/update overhead is significantly increased. In contrast, our previous work, LHT [12], fills internal nodes with data by an elegant mapping mechanism and achieves high query efficiency without compromising the index maintenance efficiency. Nevertheless, LHT can deal with one-dimensional data only. Also, the load balancing issue is not well addressed in LHT. As a non-trivial extension, m-LIGHT employs a novel multi-dimensional naming mechanism to index multi-dimensional data. Moreover, a data-aware index splitting strategy is proposed in m-LIGHT to address the load imbalance problem aggravated by multi-dimensional space partitioning.

2.2. Multi-dimensional Query Processing

In the presence of various one-dimensional P2P indexes, there are generally three solutions to processing multi-dimensional queries. The first is to employ multiple independent indexes with each indexing one attribute/dimension. Mercury [13] uses a multiple-ring structure (equivalently, multi-Chord), and processes range queries across the multiple indexes in parallel. This solution typically amplifies the index maintenance overhead and query bandwidth. The second solution is SFC indexing, which uses the Space Filling Curve to reduce dimensionality and indexes data by one-dimensional P2P indexes [4], [14], [15]. Specifically, PHT [4] applies SFC indexing over generic DHTs, while SCRAP [15] and Squid [14] apply SFC indexing to Skip graphs and Chord overlay, respectively. But the problem in SFC indexing is that the neighborhood in a multi-dimensional space is not well preserved in the one-dimensional SFC space, thus deteriorating query efficiency. The last solution is to directly develop multi-dimensional indexes, which conventional multi-dimensional indexes (e.g., kd-tree) are used to index data and mapped into P2P networks. MURK [15] and SkipIndex [16] both extend the one-dimensional Skip graphs by incorporating the kd-tree index. However, these two schemes are only applicable to some specific P2P networks. Distributed Quad-Tree [17] and DST [18] superimpose the quad-tree over DHTs and respectively support spatial queries and multi-dimensional range/cover queries. Instead of employing the quad-tree, our proposed m-LIGHT superimposes the kd-tree over DHTs. Compared to the quad-tree, the kd-tree is more flexible in space partitioning and attains better load balance. Furthermore, the kd-tree is essentially a binary tree, which, as will be seen, is suitable for incremental maintenance of m-LIGHT.

3. m-LIGHT Indexing Scheme

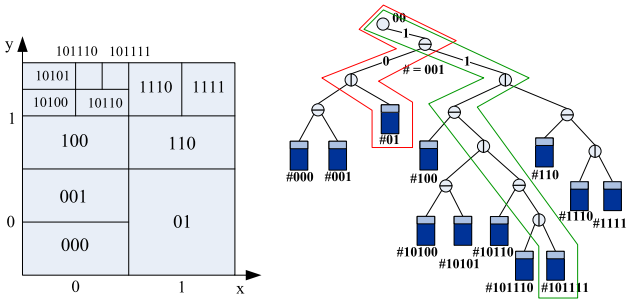
In this section, we describe the m-LIGHT index structure and its mapping strategy to the underlying DHT.

3.1. Overview

Consider a set of *data records*. Each record has a *data key* (denoted by δ), which is represented by a multi-dimensional vector $\delta = <\delta_1, \delta_2, \cdots, \delta_m>$. Without loss of generality, we assume that each δ_i $(1 \leq i \leq m)$ is a real number in interval [0, 1].

To assign data records in the underlying DHT space, each record needs a *DHT key* (denoted by κ). Given a DHT key κ, the record is mapped to the peer whose identifier is less than but closest to $hash(\kappa)$. One can simply set the data key as the DHT key, which however destroys data locality and impedes effective range query processing. Instead, m-LIGHT uses a novel method to generate DHT keys that preserve data locality. First, data keys are clustered in a *space kd-tree*, which is then decomposed into a set of distributed data structures, called *leaf buckets*. After that, a DHT key is generated for each leaf bucket by an innovative *m-dimensional naming function* such that neighboring index nodes can be easily located in distributed query processing and minimal maintenance is required for data updates. In what follows, we detail each of these procedures.

3.2. Indexing in Space Kd-Tree



(a) Space partitioning (b) Space kd-tree decomposition

Figure 1: Space kd-tree

In order to index multi-dimensional data, we recursively partition the data space into *cells* along different dimensions in an alternative fashion. As shown in Fig. 1a, the 2D space is recursively halved along the x and y axes, alternatively, until a cell contains no more than θ_{split} data records. Space partitioning is used here, that is, a data space is always equally partitioned, regardless of the data distribution. This space partitioning approach renders the local space indexed by each node to be known globally, which is essential to support distributed query processing. The index is called *space kd-tree*, as shown in Fig. 1b; every internal node has two children and the 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 non-leaf nodes. As will be discussed later, this property enables us to name each leaf node with a distinct internal node.

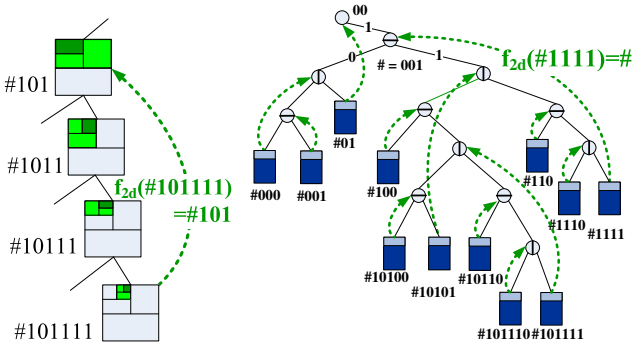
Every tree node is tagged with a label. In particular, the virtual root is labelled with $0\ldots0$ (m consecutive 0's, where m is the data dimensionality) and the ordinary root is labelled with $0\ldots01$, denoted by $\#$ (i.e., $\# = 0\ldots01$). Every tree edge is also tagged — if the edge goes left, it is labelled with 0; otherwise, 1. Then, the label of each internal node or leaf node can be obtained by concatenating all labels on the path from the virtual root to the node itself, as illustrated in Fig. 1b.

3.3. Index Decomposition

To materialize the tree in a distributed setting, we decompose the space kd-tree and store each piece in a *leaf bucket*. Conceptually, we decompose the global index tree into local trees, each of which is associated with a distinct leaf. The local tree of a leaf consists of all its ancestors. For example, Fig. 1b illustrates two local trees of leaves $\#01$ and $\#101111$. With our node labelling strategy, each local tree can be encoded in the corresponding leaf label λ: the label of each ancestor is a prefix of λ, and the sibling of an ancestor (called *branch node*) can be found by a modified prefix of λ with the ending bit inverted (i.e., 0 to 1, or 1 to 0). Thus, in a leaf bucket, we store two components: the *label store* which maintains label λ and summaries the local tree information, and *record store* which keeps all related data records. Now that the space kd-tree is decomposed into leaf buckets, the remaining issue is how to map them to the peers, which is achieved by an innovative naming function.

3.4. m-Dimensional Naming Function

For a leaf bucket labelled as 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 in the DHT peer that is responsible for $hash(f_{md}(\lambda))$. In this section, we first present the naming function for 2D indexing, and then extend it to m-dimensional indexing.



(a) Naming function (b) Bijective mapping from leaves to internal nodes

Figure 2: Naming the space kd-tree in m-LIGHT

3.4.1. Naming for 2D Indexing.

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

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

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 third last 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 $f_{2d}(\#101111) = \#101$. In particular, $f_{2d}(\#) = f_{2d}(001) = 00$. Intuitively, this naming function maps a leaf node to the lowest ancestor that is not aligned with the leaf node in terms of the quadrant position. For example, as shown in Fig. 2a, leaf node $\#101111$ lies in the top-right quadrant of its grandparent. The direct parent, node $\#10111$, also lies in the top-right quadrant of its own grandparent, so does

node $\#1011$. Thus, the naming function passes all these ancestors, until the ancestor $\#101$ is found, which is in the top-left quadrant of its own grandparent.

The naming function $f_{2d}(\cdot)$ has several interesting properties, which are described in the following theorems.[1]

*Theorem 1 (***Corner preservation):** Given an internal node ω whose corresponding data region has four corner cells, these cells are named to $f_{2d}(\omega), \omega, \omega 0$ and $\omega 1$, respectively.

Theorem 1 implies that given ω, the names of its four corner cells can be directly inferred, which is especially useful for processing of distributed range queries (since it helps to quickly locate the range boundaries).

*Theorem 2 (***Bijective mapping):** $f_{2d}(\cdot)$ is a bijective mapping from Λ to Ω, where Λ and Ω denote the leaf node set and the internal node set, respectively.

Fig. 2b shows the intuition for Theorem 2. This theorem guarantees that for each DHT key (i.e., the label of an internal node), there is one and only one leaf named to it, implying the storage load is balanced.

3.4.2. Scale up to m-dimensional Indexing.

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

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

Theorem 3 (m**-dimensional corner preservation):** In the m-dimensional index tree, given any internal node ω whose corresponding data cube has 2^m corner cells, these cells are named to $f_{md}(\omega), \omega, \omega 0, \omega 1, \omega 00, \omega 01, \cdots$, and $\omega 11 \cdots 1$, respectively.

Theorem 4 (m**-dimensional bijective mapping):** $f_{md}(\cdot)$ is a bijective mapping from Λ to Ω, where Λ and Ω denote the leaf node set and the internal node set, respectively.

In the rest of this paper, for simplicity our discussions are based on a 2D space. Nevertheless, all the algorithms presented can be extended to an m-dimensional space in a natural way.

4. Index Tree Maintenance

In this section, we discuss how m-LIGHT adjusts its structure along with data insertions and deletions.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 index splitting strategy which offers optimal load balance among peers.

1. Due to space limitations, all theorem proofs are omitted in this paper, and can be found in our technical report [19].

4.1. Incremental Tree Maintenance

In the conventional threshold-based splitting strategy, two thresholds, namely θ_{split} and θ_{merge}, are predefined for leaf split and merge. After a data insertion, if the number of records stored in the leaf bucket gets higher than θ_{split}, a split process is triggered. Similarly, after a data deletion, if a pair of sibling leaf buckets is found containing less than θ_{merge} data records, a leaf merge is then triggered. For split/merge consistency, θ_{merge} is set smaller than θ_{split} (e.g., $\theta_{merge} = \theta_{split}/2$). Before introducing the split/merge process, we present a property of our 2D-naming function.

*Theorem 5 (***Incremental split):** Consider a leaf bucket λ that is split into two child nodes, $\lambda 0$ and $\lambda 1$. The naming function $f_{2d}(\cdot)$ maps one child 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 re-assign the bucket named to λ in the underlying DHT space. For the one named to $f_{2d}(\lambda)$, it is mapped to the same peer as does bucket λ and incurs no transfer. Similarly, to merge a pair of leaf buckets, only one bucket needs to be transferred across the DHT. This nice property, termed as *incremental tree maintenance*, typically reduces the cost by half for both the number of DHT-lookups and the amount of transferred data.

4.2. Data-aware Splitting Strategy

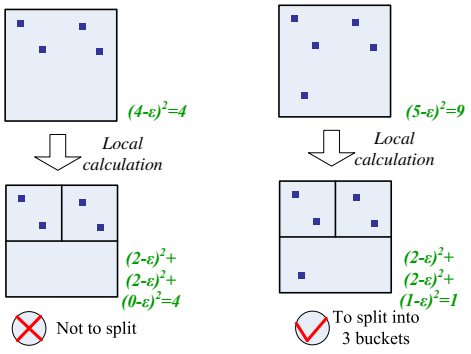
We observe that the threshold-based splitting strategy may generate empty leaf buckets, since the space-based partition employed the kd-tree does not take into account the local data distribution. In this section, we propose a data-aware index splitting strategy which achieves optimal load balance among peer nodes.

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 on each bucket. Generally, this strategy aims at minimizing the difference between the real load and the expected load (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 on the bucket. For example, in Fig. 3a, each point represents a data record (or a data key) in the data space, and $\epsilon = 2$. The optimal split subtree (shown in the bottom part) contains three leaves (or data cells), and the total difference is $(2-\epsilon)^2 + (2-\epsilon)^2 + (0-\epsilon)^2 = 4$. This

value is minimized; for instance, if the upper-left cell is further split into two leaves (respectively containing one data point), the total difference would be $(1-\epsilon)^2+(1-\epsilon)^2+(2-\epsilon)^2+(0-\epsilon)^2=6$, which is larger than the previous one, 4. To find out the minimized total difference, a naive solution is to apply the brutal-force search to try all possibilities, which is time-consuming. Instead, we use a divide-and-conquer approach, as shown in Algorithm 1 — 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 (i.e., the one for the current bucket without splitting). 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 whenever the bucket load changes (due to data insertions/deletions).

Algorithm 1 local-split(leaf bucket λ)

```
1:  s_local ← (λ.load − ε)^2
2:  if λ.load ≤ ε then
3:      return s_local.
4:  else
5:      s_left ← local-split(λ.leftChild())
6:      s_right ← local-split(λ.rightChild())
7:      s_non_local ← s_left + s_right
8:      if s_local ≤ s_non_local then
9:          return s_local.
10:     else
11:         return s_non_local.
```



(a) Before insertion (b) After insertion

Figure 3: An example for data-aware splitting ($\epsilon=2$).

An example. As shown in Fig. 3a, the leaf bucket initially contains 4 data points. Given $\epsilon=2$, the initial difference value is $(4-\epsilon)^2=4$. It then locally computes the optimal split subtree, which partitions the space into 3 cells, which contain 2, 2, and 0 data points, respectively. The minimized difference becomes $(2-\epsilon)^2+(2-\epsilon)^2+(0-\epsilon)^2=4$. Since this difference value actually equals the initial one, the split process would not be triggered. Now suppose that a new data point (0.2, 0.2) is inserted (see Fig. 3b). In this case, the initial difference value will be updated to $(5-\epsilon)^2=9$, and the minimized difference value will be $(2-\epsilon)^2+(2-\epsilon)^2+(1-\epsilon)^2=1$. Therefore, the minimized difference is smaller and, hence, the leaf bucket is split into 3 buckets, corresponding to the 3 cells shown in the bottom part of Figure 3b.

The following theorem shows that the proposed data-aware splitting strategy can achieve optimal peer load balance.

*Theorem 6 (***Optimal balance):** For a given data set and an expected number of buckets, the data-aware index splitting strategy minimizes the variance of expected load on all DHT peers.

5. Lookup Operation

Given a data key δ, the m-LIGHT lookup operation[2] returns the label of the leaf bucket that covers δ, namely $\lambda(\delta)$. The lookup operation is fundamental for supporting many other m-LIGHT operations, including exact-match queries, data insertions/deletions and range queries.

To conduct a lookup operation for δ, a peer first locally calculates the set of all possible values of $\lambda(\delta)$, called the candidate set. For example, given $\delta=<0.2,0.4>$, the binary representations of 0.2 and 0.4 are $001\cdots$ and $011\cdots$, respectively. These two binary numbers are then interleaved as $001011\cdots$, and the target label $\lambda(<0.2,0.4>)$ must be a prefix of $\#001011\cdots$. For example, in Fig. 1a, $\lambda(<0.2,0.4>)=\#001$. Furthermore, we assume that the maximum possible height of the index tree is known in advance, denoted by D, which can can be estimated by apriori knowledge or by probing certain values before query processing [8], [11]. Thus, the target label $\lambda(\delta)$ has a length in the range from 3 to $D+3$ (recall that root label $\#$ has 3 bits). As such, the lookup problem becomes how to find the target label from a candidate set of $D+1$ labels, each being a distinct prefix of the longest label.

To efficiently resolve the lookup problem, m-LIGHT employs a binary search procedure. Specifically, in each loop iteration, the algorithm first obtains a label with length being the middle value of a binary-search interval, and then applies the naming function to this label to get a DHT key and probe the corresponding peer/bucket. The lookup process is illustrated by the following example.

An example. Consider a lookup of $<0.3,0.9>$ with $D=20$. As shown in Fig. 1, the target bucket is cell $\#101110$. Note that the longest candidate label of $<0.3,0.9>$ is $\#10111000011110000111$. The m-LIGHT lookup algorithm first probes the prefix of half length, $\#1011100001$, and performs a DHT-lookup for

2. In this paper, we refer to "m-LIGHT lookup" as "lookup" for short, and as a distinction, "DHT-lookup" retains its full name.

$f_{2d}(\#1011100001) = \#101110000$. It returns a $NULL$ value and the upper search bound is decreased to $\#101110000$. The next probe is $f_{2d}(\#10111) = \#101$. The returned bucket is $\#101111$, which does not contain $<0.3, 0.9>$. Note that this probe has also examined candidate label $\#1011$, since it is also named to $\#101$. The next probe is $f_{2d}(\#101110) = \#0111$, which reaches the target $\#101110$.

6. Range Queries

In a multi-dimensional space, a range query specifies a multi-dimensional 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.

Consider a queried range R issued by some user. The peer node where the query is received from the user, called the *query initiator*, first locally figures out the lowest internal node that fully covers R (a.k.a., the *lowest common ancestor* (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 of 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 in the local tree, there exist one or more whose regions overlap the queried range. Denote these branch nodes by $\beta_1, \beta_2, \cdots$ and β_k, 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, that is, $R_i = \beta_i \cap R$. 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 approach employed in m-LIGHT. Hence, the subqueries R_i $(i = 1, 2, \cdots, k)$ can be processed in parallel and there is no redundant bucket visit. For further forwarding in each β_i, a similar process is recursively applied until the current R is fully covered in one cell. Algorithms 2 and 3 formally describe the range query processing.

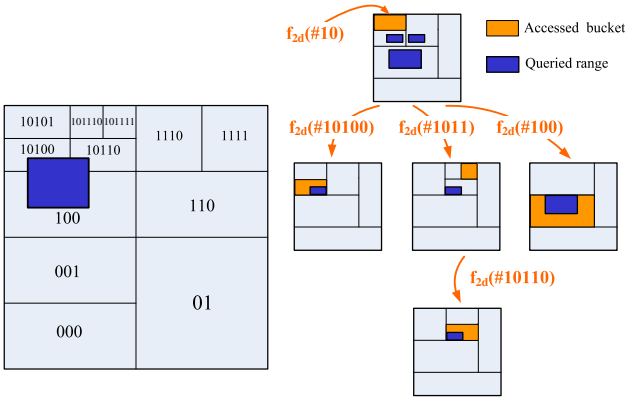
Algorithm 2 range-query(range R)

```
1: ω_R ← lowest-common-ancestor(R)
2: λ ← DHT-lookup(f_md(ω_R))
3: if λ == NULL then
4:     return lookup(R.top_left_corner)
5: else if R ⊆ λ then
6:     return λ
7: else
8:     return recursive-forward(R, ω_R)
```

An example. Suppose that the queried range is a rectangle R bounded by $[0.1, 0.3]$ in the x dimension and $[0.6, 0.8]$ in the y dimension, and that the indexed space

Algorithm 3 recursive-forward(range R, region β)

```
1: λ ← DHT-lookup(f_md(β))
2: for all β_i ∈ {branch nodes between λ and β} do
3:     R_i ← β_i ∩ R
4:     if R_i ≠ NULL then
5:         recursive-forward(R_i, β_i)
```



(a) Exemplar queried range (b) Query processing

Figure 4: Range query processing

in the current m-LIGHT is as shown in Fig. 4a. The peer receiving R computes the LCA being $\#10$ and forwards the query to the DHT key $f_{2d}(\#10) = \#1$. It is the cell with label $\#10101$ that is named to $\#1$, so is the one forwarded to. Based on the local tree of $\#10101$, the queried range is decomposed into three subranges, which are forwarded to $f_{2d}(\#10100)$, $f_{2d}(\#1011)$ and $f_{2d}(\#100)$, respectively, as illustrated in Fig. 4b. The subranges in $\#10100$ and $\#100$ are fully covered in the next peers. For the subrange in $\#1011$, the next peer is the cell $\#101111$ (note $f_{2d}(\#101111) = f_{2d}(\#1011)$), which does not cover the subrange. The query is then forwarded to $f_{2d}(\#10110)$, which covers the subrange and the process is terminated. The whole querying process consumes four DHT-lookups (in bandwidth) and three rounds of DHT-lookups (in latency).

We further develop a parallel version of range query processing. The idea is to forward h subqueries ($h >= 2$) within a branch node in each step (if $h = 1$, the parallel query processing will be degraded to the basic algorithm as previously described). By query parallelization, this processing strategy reduces latency by a factor of $h + 1$, while incurring more bandwidth as a trade-off. In practice, the user can tune the parameter of h based on his/her performance preference.

7. Performance Evaluation

This section presents the results of performance evaluation. Note that m-LIGHT is a multi-dimensional over-DHT index. Thus, we compare it with the state-of-the-art

schemes in the same category, i.e., PHT [4] and DST [5], [18]. The performance metrics of our interest are index maintenance overhead, load balance and query cost.

7.1. Experiment Setup

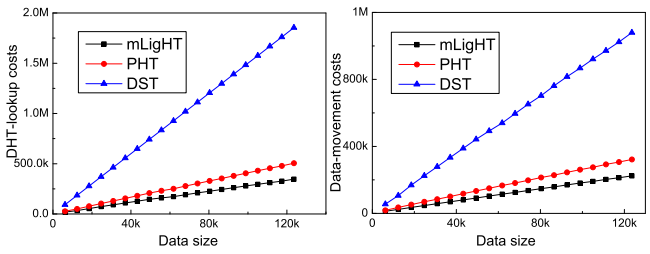
We have implemented the m-LIGHT index in Java. The total number of code lines is about 2500 (including m-LIGHT, DST and PHT), which demonstrates the simplicity of developing an over-DHT indexing scheme. In the experiments, m-LIGHT, DST, and PHT were run over the Bamboo DHT [20], a ring-like DHT that has good robustness and is now deployed in a real-life project, OpenDHT [10]. Our experimental study is based on a system built in a LAN environment where runs more than one hundred logical peers. Our experiments are based on a real dataset [21] that 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 the performance under different dataset sizes.

7.2. Maintenance Performance

The first experiment evaluates the index maintenance performance 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, i.e., the DHT-lookup cost and the data-movement cost. The results are shown in Figs. 5a and 5b. For all three indexing schemes under comparison, the cumulative maintenance costs go up linearly as data are inserted. We also vary the threshold θ_{split} and report the evaluation results in Figs. 5c and 5d. In general, both of the 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 at these nodes, thereby decreasing the data-movement cost. Comparing the three indexing schemes, due to data replication, DST is worse than the other two by an order of magnitude; m-LIGHT achieves the best performance in all cases tested and saves about 40% maintenance cost against PHT.

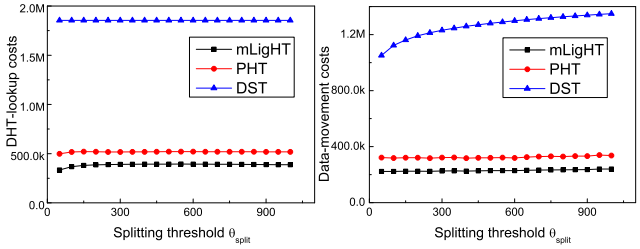
7.3. Effect of Data-aware Splitting

We now evaluate the effect of the data-aware splitting strategy in terms of load balance. We use two measures, i.e., the variance of storage on each peer and the percentage of the empty buckets. We compare it with the threshold-based splitting, which is commonly used in many existing P2P indexes. We set ϵ and θ_{split} respectively at 70 and 100, in which case the two trees under comparison are



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

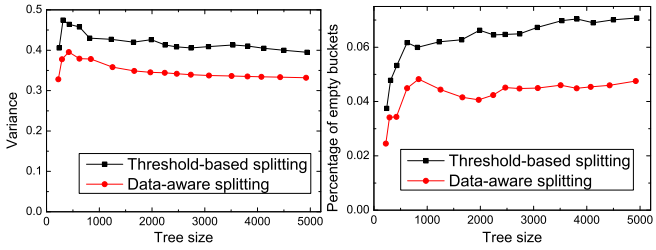
Varying data size



(c) DHT-lookup costs (d) Data-movement costs

Varying θ_{split}

Figure 5: Maintenance costs

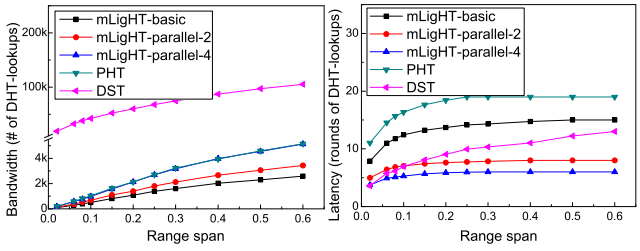


(a) Load variance (b) Percentage of empty buckets

Figure 6: Storage load balance

of the same size. The results are shown in Fig. 6. It can be seen that by the data-aware splitting strategy, the load variance is decreased by 15%, and the empty buckets are reduced by 35%.

7.4. Range Query Performance



(a) Bandwidth (b) Latency

Figure 7: Range query performance

We evaluate the range query performance in terms of bandwidth cost and response latency. The two measures used are the number of DHT-lookups and the rounds of

DHT-lookups. In evaluation, we include both the basic algorithm and the parallel algorithm for range query processing in 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 together with PHT and DST. In the experiments, the queried ranges are rectangles uniformly distributed in the data space of $[0 \dots 1, 0 \dots 1]$. We first vary the range span (i.e., the area of the rectangle) and report the results in Figs. 7a and 7b. In terms of bandwidth cost (Fig. 7a), DST consumes much more than any other scheme, typically by an order of magnitude. This is partly because in our setting, D=28; this is larger than the real tree depth, rendering the queried range to be decomposed into many small subranges in DST. In contrast, m-LIGHT (basic) is most bandwidth-efficient. The m-LIGHT (parallel-2) and m-LIGHT (parallel-4) consume more in bandwidth, but as a trade-off, they achieve a significant saving in query latency (see Fig. 7b). DST is time-efficient when the query range is small. However, as the query range increases, the latency of DST dramatically increases, whereas the other schemes are more stable.

In summary, the proposed m-LIGHT is more flexible and outperforms PHT and DST in terms of both index maintenance and query processing. Moreover, m-LIGHT (parallel) trades bandwidth efficiency for significant saving in query latency.

8. Conclusion

This paper has proposed m-LIGHT, a low-maintenance yet query-efficient multi-dimensional index structure over DHTs. Three core techniques contribute to the efficiency of m-LIGHT: a tree-decomposition strategy, a novel naming mechanism and a data-aware index splitting strategy. Experimental results based on a real dataset show that m-LIGHT outperforms the state-of-the-art schemes in various aspects, including maintenance efficiency, load balance and range query performance. As an over-DHT indexing scheme, m-LIGHT is adaptable to any DHT substrate, and is easy to implement and deploy.

Acknowledgement

This work was supported by National Natural Science Foundation of China under grant no. 90612007 and 60873070, 863 Program grant 2009AA01Z135, and Shanghai Leading Academic Discipline Project No. B114. Shuigeng Zhou was also supported by K.C. Wong Education Foundation-HKBU. Jianliang Xu was supported by grants HKBU210808 and HKBU211307 from RGC of Hong Kong. Wang-Chien Lee was supported by grants IIS-0534343 and CNS-0626709 from NSF.

References

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

[2] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, 2001, pp. 161–172.

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

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. M. Hellerstein, "A case study in building layered DHT applications," in *SIGCOMM*, 2005.

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

[6] J. Aspnes and G. Shah, "Skip graphs," in *SODA*, 2003.

[7] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "BATON: A balanced tree structure for peer-to-peer networks," in *VLDB*, 2005, pp. 661–672.

[8] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Brief announcement: prefix hash tree," in *PODC*, 2004, p. 368.

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

[10] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public DHT service and its uses," in *SIGCOMM*, 2005, pp. 73–84.

[11] P. Yalagandu and J. Browne, "Solving range queries in a distributed system," *Tech. Rep. TR-04-18, UT CS*, 2003.

[12] Y. Tang and S. Zhou, "LHT: A low-maintenance indexing scheme over DHTs," in *ICDCS*, 2008, pp. 141–151.

[13] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," in *SIGCOMM*, 2004, pp. 353–366.

[14] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *HPDC*, 2003.

[15] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: Multidimensional queries in P2P systems," in *WebDB*, 2004, pp. 19–24.

[16] C. Zhang, A. Krishnamurthy, and R. Y. Wang, "Brushwood: Distributed trees in peer-to-peer systems," in *IPTPS*, 2005.

[17] E. Tanin, A. Harwood, and H. Samet, "Using a distributed quadtree index in peer-to-peer networks," *VLDB J.*, vol. 16, no. 2, pp. 165–178, 2007.

[18] 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.

[19] Y. Tang, S. Zhou, J. Xu, and W. Lee, "A lightweight multi-dimensional index for range queries over DHTs," *Technical Report, School of Computer Science, Fudan University*, 2008. [Online]. Available: http://admis.fudan.edu.cn/member/yztang/mlight-tr.pdf

[20] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *USENIX*, 2004.

[21] "North east dataset," in *http://www.rtreeportal.org/datasets/spatial/US/NE.zip*.