Modified Skip Graph for world wide range query searching using multidimensional indexing

Upinder Kaur,

Student,

Dr. Pushpa Rani Suri,

Professor,

Department of Computer Science and Application,

Kurukshetra University,Kurukshetra

Abstract:

A skip graph is a resilient application-layer routing structure that supports range queries of distributed multidimensional data. By sorting deterministic keys into groups based on locally computed random membership vectors, nodes in a standard skip graph can optimize range query performance in distributed networks. We propose a modified skip graph, skip graph with superimposed kd index tree for multidimensional dimensional data and sorting within groups is based on locally computed random keys Compared to the state-of-the-art indexing schemes, m-ary substantially saves the index maintenance overhead, achieves a more balanced load distribution, and improves the range query performance in both cost and response latency in different environments like distributed networks, cloud networks and social networks etc.

Index terms: Skip list, Skip Graph, range queries and distributed networks.

I.INTRODUCTION

The handling of the huge amount of multidimensional data in a scalable distributed decentralized architecture is a major challenge in the era of distributed computing [1–5]. Nowadays, the applications of distributed computing are diverse- in various fields like- P2P social networks [6], service discovery in cloud computing [7], P2P e-learning systems [8], mobile P2P networks [9], massively multiplayer interactive gaming [10],[11],overlay networks etc. However, these distributed applications share larger amount of the multidimensional data distributed among thousands of the peers and hence the searching of the objects in the specific range (range query), is widely preferred as compared to the exact match queries. For example the algorithm for massively multiplayer games needs to explore various game scenarios in the specified range to identify the players and their roles in the virtual game world

[10], [11]. Therefore, the distributed networks supporting only one dimensional objects and exact match queries are inefficient to handle these applications effectively. The working of these systems are generally divided into two parts- application constraints (maps data space to identifier space and execute the search queries) and system constraints (organizes the peers and maintains the network topology) [12]. distributed networks with application constraints based on the hierarchical tree structures viz. P-tree [13], BATON [14], DP-tree [15], DHR-tree [16], VBI-tree [17], BATON* [18], CAN-QTree [19], EZSearch [20], SDItree [21], P2PM-tree [22], AVL MI-tree [23], HD-tree [12] have been proposed in the existing literature. However, as per our literature survey, the current state of the art representing distributed networks with the tree indexing suffer from the following issues related to the application constraints and/or system constraints, limiting the range query search in the distributed networks like p2p:

• The application constraints have limited support to the range query of the multidimensional data, i.e. the P2P tree indexing supports range query for only one-dimensional data [13–15], [18]. However, the above mentioned applications involve the support of the multidimensional range queries in a distributed dynamic environment.

• Even though the P2P tree indexing is extended for the multidimensional range query, the operations of the system constraints have limited bounds. The range query search bound is generally limited to - O(log2N) for binary trees [14], [17], [21], [23] or O(N1/2) [19] or dimensionality impacts the range query search performance [20], [22].

• In most of the P2P tree indexing, the same node is responsible to handle the operations of the application constraints and the system constraints, i.e. the node acts as the routing node (to direct the peers and the rage queries in the topology) as well as the data node (to store the index of the data objects) [13], [14], [18], [19], [22]. However, the efficiency of the P2P tree indexing can be increased by assigning the routing operations and the data indexing and query search operations (like range query) to the different nodes. Based on the above observations, we believe a P2P hybrid topology can be developed that makes use of the multiway tree (fanout m > 2) to reduce the cost of the complex range query search operations and that also supports the Multidimensional Indexing (MI) to handle the above mentioned applications.

Our contributions in this paper are-

• The illustrative modified model that extends the skip graph with kd [18] that reduces the range query search cost from O(log2N) to the O(logmN) (m is the fanout of the tree, m > 2). In addition, it also supports the MI based on the space containment relationships such as R-tree [24]orSS-tree[25].
• However, our focus in the paper is the range query search algorithm in this hybrid model. We carry out numerous experiments to analyze the cost of the range query search. Through our results we show that the cost of the range query search is bound to O(logmN). We also show that

the cost of the range query search is independent of the dimensionality of multidimensional objects.

1.1. SKIP GRAPH

A Skip Graph is a distributed data structure for various distributed applications to search for keys. Each node in Skip Graph has two fields: key and membership vector. Let *m* (*p*) denote the membership vector of node *p*. The elements of *m* (*p*) belong to a finite alphabet set Σ. We denote by *b* the size of the alphabet; i.e., / Σ / = *b*. We think of *m* (*p*) as an infinite word over Σ; but in practice, only an O (log *N*) length prefix of *m* (*x*) is needed on average. There are multiple levels in a Skip Graph, and nodes are grouped into increasingly smaller doubly lists within each successively higher level. The levels are ranging from level 0. In every list, peers are lexicographically sorted in their keys. At level 0, all peers belong to one list. At level 1, all peers are separated into *b* lists. Similarly, all peers in a list of level *I* are separated into *b* lists at level *i* +1. The membership vector determines which lists the peer belongs to at each level. Two peers *p* and *q* belong to the same list at level *i* iff *m*(*p*) and *m*(*q*) have the identical prefix of length *i*. Figure 1 shows an example of a Skip Graph with *b* = 2, which is the most common case in practice. In the figure,

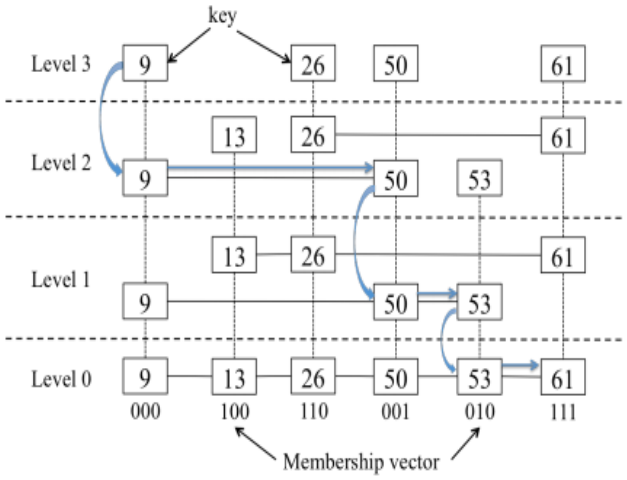


Figure. 1. A simple example of Skip Graph

the rectangles represent nodes, and the numbers inside the rectangles indicate the node's key value. The binary sequence below each node indicates the node's membership vector, the prefix of which are being used to group nodes into lists on a given level. The retrieval of an individual key is carried out downward from the higher to lower levels. This is because the links at higher levels indicate larger distances between keys than that at the lower levels. This enables efficient forwarding of search queries. At each node, the value of the query is compared to the node's key in the level where the query has arrived, and a decision is made as to whether the query is to be sent to the left or right. Then it is compared to the key of the next node. If it does not exceed the value of the query, the query is sent to

that node. If the key of the next node exceeds the value of the query, the level is decremented by 1 and the process of comparing with the keys of neighboring nodes continues until the condition is satisfied. In Figure 1, for instance, consider that we aim to retrieve key 61 from the leftmost node which holds key 9. As in this example, the keys are unique, we refer to nodes with the keys they are holding for convenience. As key 61 is larger than 9, therefore, the query is sent to the right. Then, we look for the destination of the retrieval at the node's highest level (Level 3). Since no link exists at level 3, the retrieval level is lowered by 1 (i.e., level 2). In level 2, the key held by the right neighbor node holding key 50 is referenced. In this case, since 50 is smaller than 61, the query is sent to the node 50. This process is repeated until node 61 is reached.

1.2 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 [26] 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 [27], [28], [29]. Specifically, PHT [27] applies SFC indexing over generic DHTs, while SCRAP [29] and Squid [28] 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 [29] and SkipIndex [30] 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 [31] and DST [32] 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 superimposes the kd-tree over skip graph. Compared to the quad-tree, the kd-tree is more flexible in space partitioning and attains better load balance.

2. PROBLEM STATEMENT

The efficient data storage and retrieval in dynamic and distributed network topologies is not limited to only one dimensional data and exact match queries. To handle the multidimensional applications in vary distributed environment with the reduced complex query cost especially range query search is a major challenge in the distributed computing. In this paper, we briefly discuss about the hybrid concept of the Multidimensional Indexing (MI) with the m-ary kd space tree topology (m > 2). The paper basically aims to reduce the range query search cost from O(log2N) to O(logmN)

3. Superimposes kd tree Indexing Scheme

In this section, we describe the superimposed kd tree index structure and its mapping strategy to the

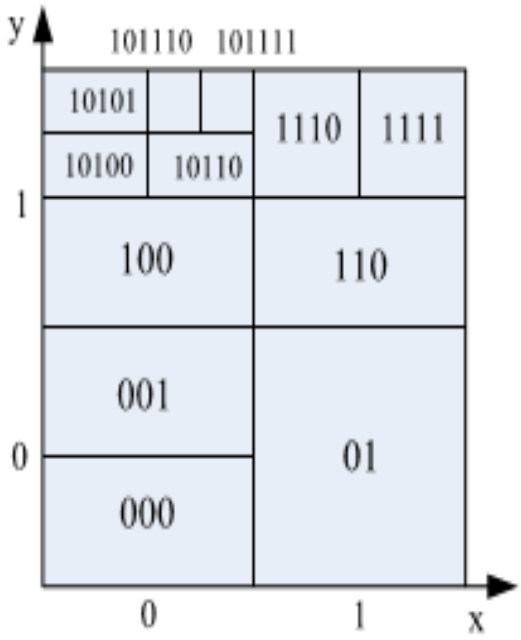
underlying skip graph.

3.1. Overview

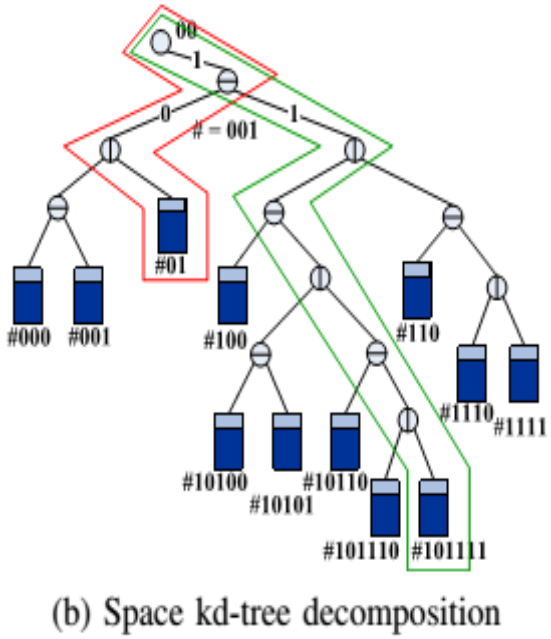
Consider a set of *data records*. Each record has a *data key* (denoted by δ), which is represented by a multidimensional vector $\delta = <\delta 1, \delta 2, \cdots, \delta m>$. Without loss of generality, we assume that each δi ($1 \leq i \leq m$) is a real number in interval [0, 1].

To assign data records in the underlying space, each record needs a *membership key* (denoted by κ). Given a membership 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 membership key, which however destroys data locality and impedes effective range query processing. Instead, uses a novel method to generate membership 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 membership 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 2: 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 Figure . 2a, the 2D space is recursively halved along the *x* and *y* axes, alternatively, until a cell contains no more than ϑ split data records. 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 Figure. 2b; 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 . . . 0 (*m* consecutive 0's, where *m* is the data dimensionality) and the ordinary root is labelled with 0 . . . 01, denoted by # (i.e., # = 0 . . . 01). 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 Figure. 2b.

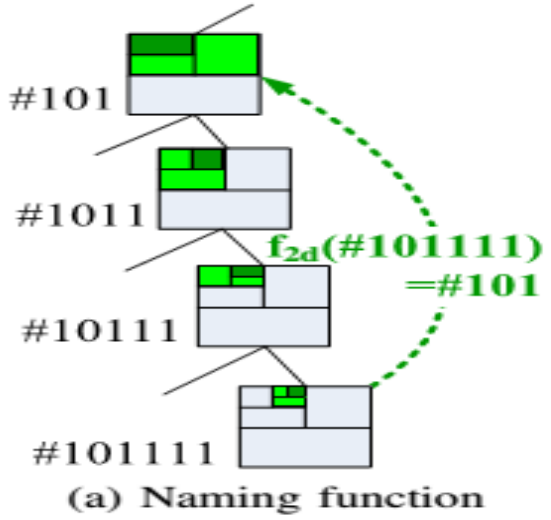
3.3. Index Decomposition

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

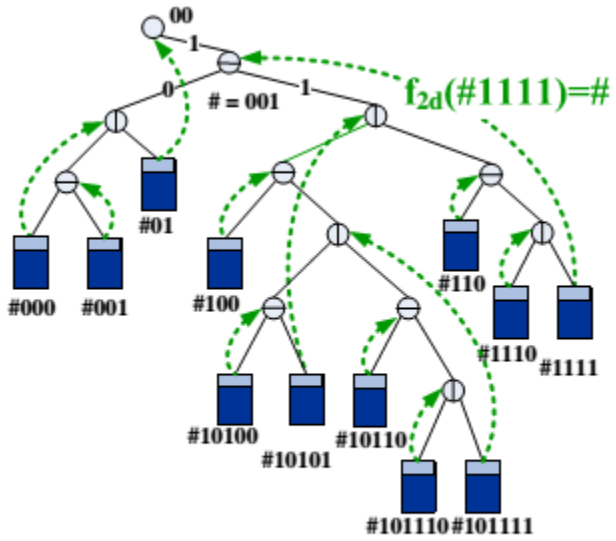
associated with a distinct leaf. The local tree of a leaf consists of all its ancestors. For example, Fig. 1b illustrates two local trees of leaves #01 and #101111. With our node labelling strategy, each local tree can be encoded in the corresponding leaf label λ: the label of each ancestor is a prefix 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 membership key κ, i.e., $\kappa = f_{md}(\lambda)$. The bucket is then stored in the skip graph peer that is responsible for $hash(_{fmd}(\lambda))$. 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 3: Naming the space kd-tree in *m*-ary

3.4.1. Naming for 2D Indexing.

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

$f_{2d}(b_1 \cdot\cdot \cdot bi-_2b_{i-1}\ b_i) = \{ f_{2d}\cdot(b_1\cdot\cdot \cdot b_{i-2}b_{i-1})$ if $b_{i-2}=b_i$, $b_1 \ldots b_{i-2}b_{i-1}$ otherwise,} Specifically, given a binary string $\lambda = b_1 \cdot\cdot \cdot b_{i-2}b_{i-1}\ b_i$, $f_{2d}(\cdot)$ 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 Figure. 3a, 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 $f_{2d}(\omega)$, ω, ω0 and ω1, 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. Figure. 3b shows the intuition for Theorem 2. This theorem guarantees that for each membership 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 $\lambda = b_1 \cdot\cdot \cdot b_{i-m} \cdot\cdot \cdot b_{i-1}b_i$, where $b_j = [0/\ 1]$, $j = 1, \cdot\cdot \cdot, i$,
the m-dimensional naming function is recursively defined by

$f_{md}(\lambda) = f_{md}(b_1 \cdot\cdot \cdot b_{i-m} \cdot\cdot b_{i-1}b_i)$

$= \{f_{md}\cdot\cdot \cdot(b_i - \cdot\cdot \cdot b_{i-m\ \ldots\ldots}b_{i-\cdots 1}$) if $b_{i-m} = b_i$, $b_{1\ldots\ldots}b_{i-m}$ ….. b_{i-1} otherwise. }

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 $f_{md}(\omega)$, *ω, ω*0*, ω*1*, ω*00*, ω*01*,* ·· · , and *ω*11 ·· · 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*-ary adjusts its structure along with data insertions and deletions.We first consider the conventional threshold-based splitting strategy and show that *m*-ary can achieve *incremental tree maintenance*..

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., $\vartheta_{merge} = \vartheta_{split}/2$). 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, *λ*0 and *λ*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 skip graph-put operation to re-assign the bucket named to λ in the underlying skip graph 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 skip graph. This nice property, termed as incremental tree maintenance, typically reduces the cost by half for both the number of skip graph-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 , 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., e). 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 $(l - e)^2$, where l is the number of data records stored on the bucket. For example, in Figure. 4a, each point represents a data record (or a data key) in the data space, and $^2 = 2$. The optimal split subtree (shown in the bottom part) contains three leaves (or data cells), and the total difference is $(2 - e)^2 + (2 - e)^2 + (0 - e)^2 = 4$. 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 $(1 - e)^2 + (1 - e)^2 + (2 - e)^2 + (0 - e)^2 = 6$, 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 e 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: slocal ← (λ.load − e) 2
2: if λ.load ≤ e then
3: return slocal.
4: else
5:      sleft ← local-split(λ.leftChild())
6:      sright ← local-split(λ.rightChild())
7:      snon local ← sleft + sright
8:  if slocal ≤ snon local then
9:       return slocal.
10:  else
11:      return snon local.
```

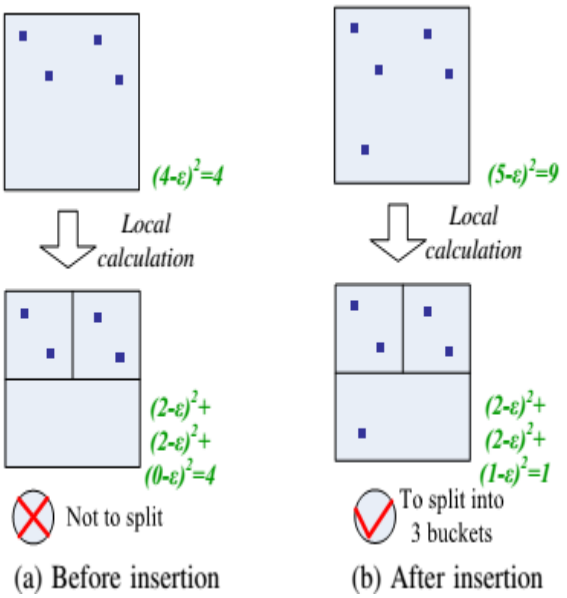


Figure 4: An example for data-aware splitting (e = 2).

An example. As shown in Figure. 4a, the leaf bucket initially contains 4 data points. Given e= 2, the initial difference value is $(4 - e)^2 = 4$. 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 $(2 - e)^2 + (2 - e)^2 + (0 - e)^2 = 4$. 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 Figure. 4b). In this case, the initial difference value will be updated to $(5 - e)^2 = 9$, and the minimized difference value will

be $(2 - e)^2 + (2 - e)^2 + (1 - e)^2 = 1$. 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 4b.

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 skip graph peers.

5. Lookup Operation

Given a data key δ, the *m*-ary lookup operation returns the label of the leaf bucket that covers δ, namely $\lambda(\delta)$. The lookup operation is fundamental for supporting many other *m*-ary 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 δ =<0.2, 0.4>, the binary representations of 0.2 and 0.4 are 001 ⋯ and 011 ⋯ , respectively. These two binary numbers are then interleaved as 001011 ⋯ , and the target label λ(<0.2, 0.4>) must be a prefix of #001011 ⋯ . For example, in Figure. 2a, λ(<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 a priori knowledge or by probing certain values before query processing. 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*-ary 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 membership 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 Figure. 2, the target bucketis cell #101110. Note that the longest candidate label of <0.3, 0.9> is #10111000011110000111. The *m*-ary lookup algorithm first probes the prefix of half length, #1011100001, and performs a skip graph-lookup for 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*-ary 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 skip graph-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 $\theta_1, \theta_2, \cdots$ and θ_k, respectively. For each θ_i, the range query is decomposed into the subrange R_i, which is the overlapped region between θ_i and *R*, that is, $R_i = \theta_i \cap R$. Then, R_i is forwarded to θ_i via a Skip Graph-lookup of $f_{2d}(\theta_i)$. Note that there is no overlap between R_i and R_j due to the space partitioning approach employed in *m*-ary. 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 θ_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

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 2 range-query(range R)

1.ωR←lowest-common-ancestor(R)

2. λ←Skip Graph Lookup(f_{md}(ωR)

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, ωR)

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

1.λ←Skip Graph-lookup(f_{md} (β))

2. for all $\beta_i \in$ {branch nodes between λand β} do

3. $R_i \leftarrow \beta_i \cap R$

4. if $R_i \neq$NULL then

5. recursive-forward (R_i, β_i)

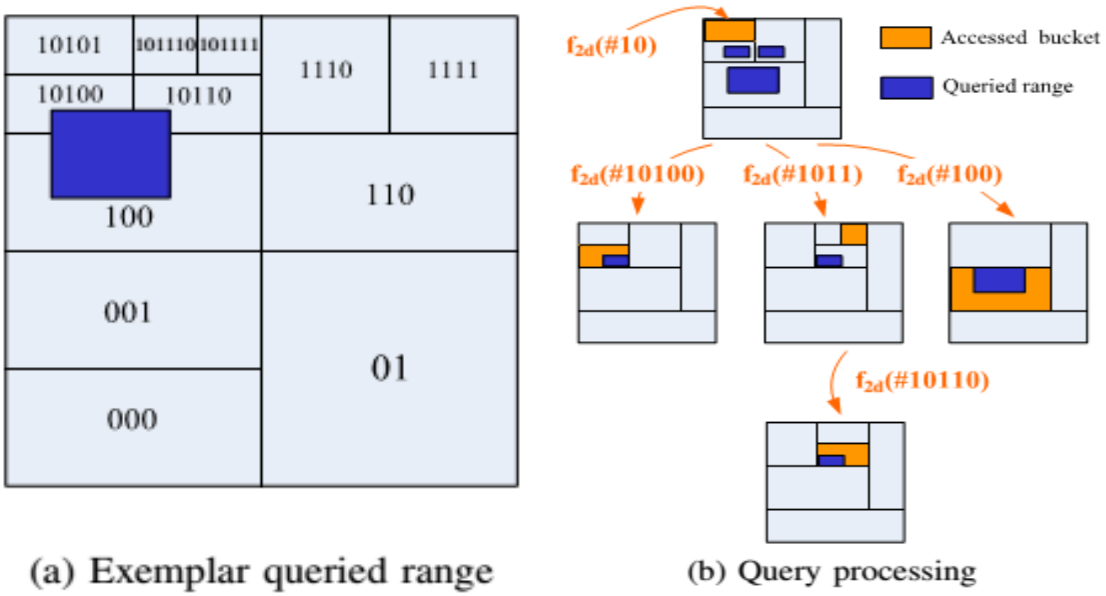


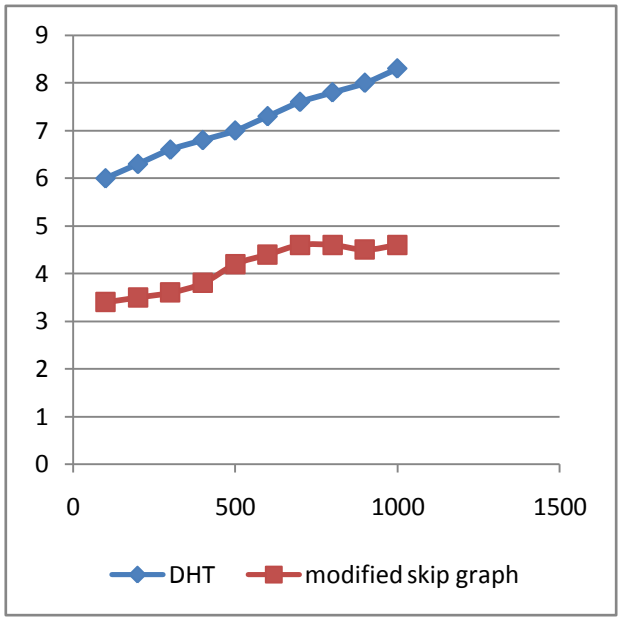
Figure 5: Range query processing

in the current *m*-ary is as shown in Figure. 5a. The peer receiving *R* computes the LCA being #10 and forwards the query to the membership key *f*2*d*(#10) = #1. It is the cell with label #10101 that is named

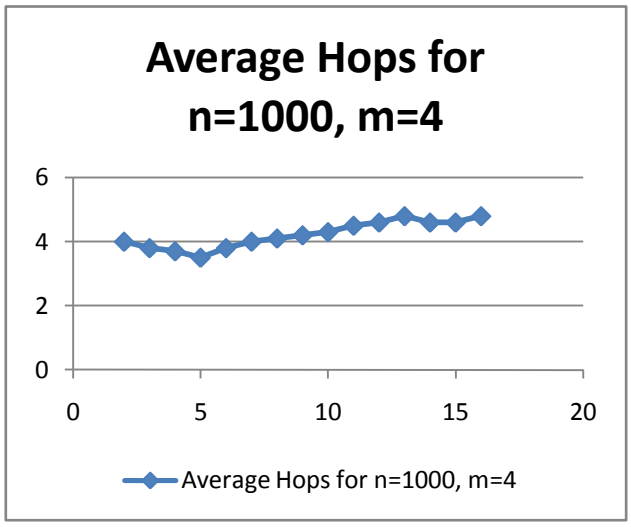
to #1, so is the one forwarded to. Based on the local tree of #10101, the queried range is decomposed into three subranges, which are forwarded to *f*2*d*(#10100), *f*2*d*(#1011) and *f*2*d*(#100), respectively, as illustrated in Figure. 5b. The subranges in #10100 and #100 are fully covered in the next peers. For the subrange in #1011, the next peer is the cell #101111 (note *f*2*d*(#101111) = *f*2*d*(#1011)), which does not cover the subrange. The query is then forwarded to *f*2*d*(#10110), which covers the subrange and the process is terminated. The whole querying process consumes four skip graph-lookups (in bandwidth) and three rounds of skip graph-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. EXPERIMENTAL STUDY

To evaluate the performance of the range query search, we implemented the modified model of skip graph using kd-tree multidimensional indexing in the peer-to-peer simulator. The range of dimension varies from 2 to 15 for experimentation and the fanout value *m*=4 is selected as estimated by the cost model of the BATON. For each experiment, 10000 uniform data objects are inserted in a network of 1000 nodes. In this setup, 100 range queries are executed for both the VBI-tree (*m*=2) and the hybrid model (*m*=4) and the average number of hops is calculated. Figure 6(a) shows the average number of hops for various network sizes (for dimension *d*=3). It can be observed that the range query in the modified model outperforms DHTs for each network size as the height of the tree is reduced in the modified model. The average number of hops for various dimensions is presented in figure 6(b) (for network size = 1000). It can be observed that the average number of hops for range query searches is independent of the dimensionality. In addition, the maximum number of hops in each experiment is limited to the height of the tree, irrespective of the number of nodes to be searched and hence it is independent of the dimensionality.



(a)



(b)

Figure. 6. Average number of hops of range query search for (a) various network sizes, (b) various dimensions

8. Conclusion and Future Scope

In this paper, we illustrated a modified model that combines skip graph and superimposed kd multidimensional indexing methods based on the space containment relationships. It gains advantages of effective storage and retrieval of multidimensional objects by reducing the search cost. The experimental results support the efficiency of this hybrid model in distributed computing.This paper also has proposed *m*-ary, a low-maintenance yet query-efficient multi-dimensional index structure for skip graphs. Three core techniques contribute to the efficiency of *m*-ary: 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*-ary outperforms the state-of-the-art schemes in various aspects, including maintenance efficiency and range query performance. As an over skip graph indexing scheme, *m*-ary is adaptable to any skip graph substrate, and is easy to implement and deploy.

In this paper, we mainly focuses on the performance of the multidimensional range query searching . We described the range query search algorithm and our analysis shows that number of hops for the range query search is bound to O(*logmN*) in the worst case as height of the *m*-ary tree is *logmN*. We experiment the multidimensional range query search algorithm for the hybrid model using P2P simulator. Our experimental results verify the superiority of the range query in the modified model as compared to the existing DHTs.

In future we will expose the skip graphs in concurrent environment to maximize benefits of concurrency in various environments.

References

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes,” Communications Surveys Tutorials, IEEE, vol. 7, no. 2, pp. 72 – 93, 2005.

[2] J. Risson and T. Moors, “Survey of research towards robust peer-to-peer networks: Search methods,” Computer Networks, vol. 50, no. 17, pp. 3485 – 3521, 2006.

[3] C. Zhang, W. Xiao, D. Tang, and J. Tang, “P2P-based multidimensional indexing methods: A survey,” Journal of Systems and Software, vol. 84, no. 12, pp. 2348 – 2362, 2011.

[4] C. Dannewitz, T. Biermann, M. Dräxler, and H. Karl, “Complex queries in P2P networks with resource-constrained devices,” Journal of Advances in Information Technology, vol. 2, no. 1, pp. 2–14, Feb. 2011.

[5] H. Bandara and A. Jayasumana, “Collaborative applications over peer-to-peer systems - challenges and solutions,” Peer-toPeer Networking and Applications, vol. 6, no. 3, pp. 257–276, 2013.

[6] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips, “Tribler: a social-based peer-to-peer system,” Concurrency and Computation: Practice and Experience, vol. 20, no. 2, pp. 127–138, 2008.

[7] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar, “Peer-to-peer cloud provisioning:

Service discovery and loadbalancing," in Cloud Computing, ser. Computer Communications and Networks. Springer London, 2010, pp. 195–217.
[8] G. Wang, Y. Yuan, Y. Sun, J. Xin, and Y. Zhang, "Peerlearning: A content-based e-learning material sharing system based on p2p network," World Wide Web, vol. 13, no. 3, pp. 275–305, 2010.
[9] L. Shou, X. Zhang, P. Wang, G. Chen, and J. Dong, "Supporting multi-dimensional queries in mobile P2P network," Information Sciences, vol. 181, no. 13, pp. 2841 – 2857, 2011.
[10] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in INFOCOM 2004: Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 1, March 2004, p. 107.
[11] Y. Hu, L. N. Bhuyan, and M. Feng, "P2P consistency support for large-scale interactive applications," Computer Networks, vol. 56, no. 6, pp. 1731 – 1744, 2012.
[12] Y. Gu and A. Boukerche, "HD tree: A novel data structure to support multi-dimensional range query for P2P networks," Journal of Parallel and Distributed Computing, vol. 71, no. 8, pp. 1111 – 1124, 2011.
[13] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying peer-to-peer networks using P-trees," in WebDB '04:Proceedings of the Seventh International Workshop on the Web and Databases, 2004, pp. 25–30.
[14] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "BATON: a balanced tree structure for peer-to-peer networks," in VLDB '05: Proceedings of the 31st international conference on Very Large Databases, 2005, pp. 661–672.
[15] M. Li, W.-C. Lee, and A. Sivasubramaniam, "DPTree: A balanced tree based indexing framework for peer-to-peer systems," in ICNP '06: Proceedings of the 14th IEEE International Conference on Network Protocols, 2006, pp. 12–21

[16] X. Wei and K. Sezaki, "DHR-Trees: A distributed multidimensional indexing structure for p2p systems," in ISPDC '06:TheFifth International Symposium on Parallel and Distributed Computing, 2006, July 2006, pp. 281 –290.
[17] H. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou, "VBI-tree: A peer-to-peer framework for supporting multidimensional indexing schemes," in ICDE '06: Proceedings of the 22nd International Conference on Data Engineering, April 2006, pp. 34–44.
[18] H. V. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. Vu, and R. Zhang, "Speeding up search in peer-to-peer networks with a multi-way tree structure," in SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2006, pp. 1–12.
[19] H. Zuo, N. Jing, Y. Deng, and L. Chen, "CAN-QTree: A distributed spatial index for peer-to-peer networks," 10th IEEE International Conference on High Performance Computing and Communications, vol. 0, pp. 250–257, 2008.
[20] D. Tran and T. Nguyen, "Hierarchical multidimensional search in peer-to-peer networks," Computer Communications, vol. 31, no. 2, pp. 346 – 357, 2008, special Issue: Foundation of Peerto-Peer

Computing.
[21] R. Zhang, W. Qian, A. Zhou, and M. Zhou, "An efficient peer-to-peer indexing tree structure for multidimensional data," Future Generation Computer Systems, vol. 25, pp. 77–88, 2009.
[22] A. Vlachou, C. Doulkeridis, and Y. Kotidis, "Peer-to-peer similarity search based on M-tree indexing," in Database Systems for Advanced Applications, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5982, pp. 269–275.
[23] L. Jin-ling and Z. Hong, "Study of the AVL-tree index range query based on P2P networks," in ICEE '10: International Conference on E-Business and E-Government, 2010, pp. 1699–1702.
[24] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in Proceedings of International conference on Management of Data. ACM press, 1984, pp. 47–57.
[25] D. White and R. Jain, "Similarity indexing with the SS-tree," in Proceedings of the Twelfth International Conference on Data Engineering, 1996, pp. 516–523.

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

[27] 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.

[28] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in HPDC, 2003.
[29] 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.

[30] C. Zhang, A. Krishnamurthy, and R. Y. Wang, "Brushwood: Distributed trees in peer-to-peer systems," in IPTPS, 2005.
[31] 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.
[32] 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