VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes

H. V. Jagadish[1,2]   Beng Chin Ooi[2]   Quang Hieu Vu[2]   Rong Zhang[3]   Aoying Zhou[3]
jag@eecs.umich.edu, ooibc@comp.nus.edu.sg, hieuvq@nus.edu.sg, {rongzh, ayzhou}@fudan.edu.cn
[1] University of Michigan, USA *   [2] NUS, Singapore †   [3] Fudan University, China †

Abstract

Multi-dimensional data indexing has received much attention in a centralized database. However, not so much work has been done on this topic in the context of Peer-to-Peer systems. In this paper, we propose a new Peer-to-Peer framework based on a balanced tree structure overlay, which can support extensible centralized mapping methods and query processing based on a variety of multi-dimensional tree structures, including R-Tree, X-Tree, SS-Tree, and M-Tree. Specifically, in a network with N nodes, our framework guarantees that point queries and range queries can be answered within $O(logN)$ hops. We also provide an effective load balancing strategy to allow nodes to balance their work load efficiently. An experimental assessment validates the practicality of our proposal.

1. Introduction

Peer-to-Peer (P2P) systems have become popular for sharing resources, particularly files, across large numbers of users. Exact match queries based on identifiers are well supported in this context. However, shared data such as documents, music files, and images, are frequently specified as points in a multi-dimensional data space based on expressed features. As a result, it is important for P2P systems to provide efficient multi-dimensional query processing.

Index structures are central to efficient search in database systems. Multi-dimensional indexes such as the R-tree [8] and R*-tree[2], and high-dimensional indexes such as the M-tree[7] have been well tested and are widely accepted as robust indexes in centralized systems. Even for P2P systems, there have been a few proposals to support multi-dimensional indexing [18, 12, 21, 19]. Most systems supporting multi-dimensional data indexing in centralized database are based on tree structures, which have many robust properties such as concurrency, scalability, and adaptivity. However, construction of such a structure is difficult in a P2P environment. As such all the P2P multi-dimensional indexing systems referenced above are based on space filling curve or space partitioning. They do not inherit the properties of well-tested multi-dimensional hierarchical indexing structures proposed in the literature.

In this paper, we propose a general framework, called the Virtual Binary Index (VBI) overlay network, based on a virtual binary balanced tree structure, which can be used to support any kind of hierarchical tree structures that has been designed based on a space containment relationship such as the R-tree and M-tree. There are two main components in the framework. The first component is the overlay network based on a balanced binary tree concept that is inspired by BATON [10]. However, the binary tree is only virtual, in that peer nodes are not physically organized in a tree structure at all. The second component defines abstract methods for multi-dimensional indexing and is extensible to a variety of index structures.

Our paper makes the following contributions

- We present a framework that is capable of supporting a variety of well-tested multi-dimensional indexing methods such as the R-Tree [8], X-Tree [4], SS-Tree [20], and M-Tree [7] in a P2P system.
- We present search algorithms for point queries and range queries, bounded in cost by $O(logN)$ hops, defined as the maximum path length of messages required to solve the problem.
- We present a comprehensive load balancing mechanism so that the framework is adaptive to data and load distributions.

The rest of the paper is organized as follows: In Section 2 we present related work. In Section 3 we provide background information about BATON. In Section 4, we introduce our framework and the overlay network. In Section 5,

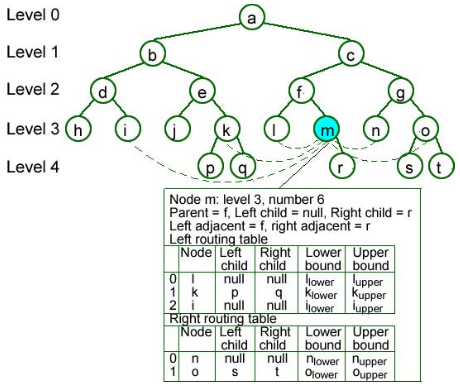
* Supported in part by NSF grant EIA-0303587
† Supported in part by IDA CCC grant as part of BestPeer [13] project

we explain our index construction and general algorithms for different indexing schemes. In Section 6 we discuss load balancing schemes used in our system. Finally, we show the performance study in Section 7 and conclude in Section 8.

2. Related Work

Multi-dimensional indexing, including high-dimensional indexing, has received extensive research attention in the context of centralized databases [5, 6]. In many of these methods, the data space is hierarchically divided into smaller subspaces (regions), such that the higher level data subspace contains the lower level subspaces and acts a guide in searching. Naturally, most such methods are tree-based. These methods can be data-partitioning based, where data subspaces are allowed to overlap (eg. R-tree) or space-partitioning based, where data subspaces are disjoint (eg. R$^+$-tree [17]). The specific partitioning technique (whether space-based or data-based) is not material to our algorithms in this paper, and hence we shall not distinguish between them.

Amongst P2P systems, CAN [14] can be considered the first system supporting multi-dimensional data, although the original intention is to hash data uniformly into multi-dimensional space such that a certain degree of fault tolerance can be guaranteed. Being a structure that has some similarity with the kd-tree[3] and grid-file[9], CAN can be used to directly index multi-dimensional data in its natural space. Most other systems such as [16, 12] use space filling curves to map multi-dimensional data to one dimensional data. After that, an overlay network is used to index that one dimensional data. These works behave poorly when the data distribution is skewed. pSearch [19], a P2P system based on CAN, is proposed for document retrieval in P2P networks by rotating the dimensions in indexing. Its focus is on retrieving relevant documents rather than on range queries. Another system also based on CAN is proposed by Sahin et al [15] in which the ranges are included into hash functions. As a result, the system can always return a superset of the range query. However, exact search is highly inefficient in this system. SkipIndex [21] is based on skip graph [1], which aims to support high dimensional similarity query. It is based on k-d-tree [3] to partition the data space, and then maps the data space into skip graph overlay network by encoding it into a unique key. However, it inherits the scaling problems of space filling curves with respect to dimensionality. Furthermore, it cannot guarantee that data is found within $O(logN)$ steps.



Node m: level 3, number 6
Parent = f, Left child = null, Right child = r
Left adjacent = f, right adjacent = r
Left routing table

	Node	Left child	Right child	Lower bound	Upper bound
0	l	null	null	l_{lower}	l_{upper}
1	k	p	q	k_{lower}	k_{upper}
2	i	null	null	i_{lower}	i_{upper}

Right routing table

	Node	Left child	Right child	Lower bound	Upper bound
0	n	null	null	n_{lower}	n_{upper}
1	o	s	t	o_{lower}	o_{upper}

Figure 1. BATON structure

3. Background - BATON

In this section, we briefly describe the recently proposed overlay network based on the binary tree called BAlanced Tree Overlay Network (BATON) [10]. An example BATON network is shown in Figure 1, where each peer in the network maintains one node of the tree. Each node in the tree is associated with a *level* and a *number*. Nodes form the network by connecting with others via links. Each node in the tree maintains "links" to its parent, children, adjacent nodes, and selected neighbor nodes. Links to selected neighbors are maintained by two sideways routing tables: a *left routing table* and a *right routing table*. Each of these routing tables contains links to nodes at the same level with numbers that are less (respectively greater) than the number of the source node by a power of 2. The definition of a balanced tree in BATON is given below for clarity:

Definition 1: *A tree is* balanced *if and only if at any node in the tree, the height of its two subtrees differ by at most one.* □

Two important theorems are central to BATON. The first theorem is used to guarantee the balance of the tree while the second gives an efficient way to forward requests among nodes in the network.

Theorem 1: *A tree is a balanced tree if every node in the tree that has a child also has both its left and right routing tables full.*[1] □

Theorem 2: *If a node x contains a link to another node y in its left or right routing tables, the parent node of x must also contain a link to the parent node of y unless the same node is parent of both x and y.* □

We note that the overlay network proposed in this paper is a balanced binary tree, but is not the same as BATON, in two crucial ways. Firstly, index nodes are mapped differently into peer nodes from BATON in which each index node corresponds to a peer. Secondly, the definition of sideways routing is new since linear ordering between value ranges at nodes can no longer be assumed as in BATON. As a result, algorithms for all operations described differ from

[1] A routing table is full if none of its valid links is NULL

those in BATON; and in particular the crucial search algorithm is completely different. Nonetheless, the two central theorems of BATON apply to the VBI-Tree we introduce below, since they share a similar binary tree structure.

4. VBI-Tree Architecture

4.1. The Overall Framework

The Virtual Binary Index Tree (VBI-Tree) is an abstract tree structure on top of an overlay network as shown in Figure 2. The abstract methods defined can support any kind of hierarchical tree indexing structures in which the region managed by a node covers all regions managed by its children. Popular multi-dimensional hierarchical indexing structures include the R-tree[8], the X-tree[4], the SS-tree[20], the M-tree[7], and their variants.

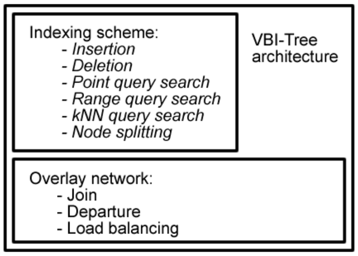
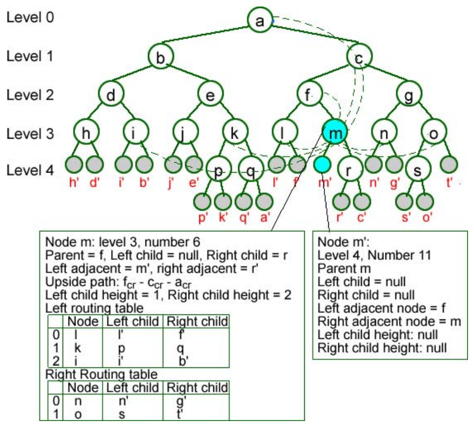


Figure 2. Overall framework

4.2. Overlay Network

As is the case for many centralized indexing methods, VBI-Tree nodes can be partitioned into two classes: data nodes (or leaf nodes) and routing nodes (or internal nodes). Data nodes actually store data while routing nodes only keep routing information. Like BATON, each routing node in the VBI-Tree maintains links to its parent, its children, its adjacent nodes and its sideways routing tables. However entries in the routing tables do not need to keep information about regions covered by neighbor nodes. Instead, each routing node maintains an "upside table", with information about regions covered by each of its ancestors. Additionally, each node needs to keep information about heights of sub-trees rooted at its children. (This is used for the network restructuring process (see 6.1)). VBI-Tree construction employs a parsimony principle which requires that every internal node have two children. The parsimony principle forces the total number of nodes in a VBI-tree to be an odd number, and exactly half of these rounded up, to be data nodes. In fact, we can establish the following theorem:



Node m: level 3, number 6
Parent = f, Left child = null, Right child = r
Left adjacent = m', right adjacent = r'
Upside path: f_{cr} - c_{cr} - a_{cr}
Left child height = 1, Right child height = 2
Left routing table

	Node	Left child	Right child
0	l	l'	f'
1	k	p	q
2	i	i'	b'

Right Routing table

	Node	Left child	Right child
0	n	n'	g'
1	o	s	t'

Node m':
Level 4, Number 11
Parent m
Left child = null
Right child = null
Left adjacent node = f
Right adjacent node = m
Left child height: null
Right child height: null

Figure 3. VBI-Tree structure

Theorem 3: *In an in-order traversal of the VBI-Tree, data nodes and routing nodes alternate.* □

Theorem 3 follows from the definition of a balanced binary tree – since any node that has a child must have full sideways routing tables, at the very least it has a sibling.

Each peer in the network is assigned a pair of VBI-Tree nodes: a routing node and a data node, in which the data node is the left adjacent node of the routing node (in the in-order traversal of the tree). The sole exception is the peer keeping the right most data node, which does not have a routing node. Since each peer keeps a routing node and a data node, and query requests can be forwarded via links in the routing node, to save space, data nodes do not need to keep sideways routing tables as well as upside path. The special peer keeping the rightmost data node, which does not have a routing node, always forwards requests to the parent node of the data node for processing. The VBI-Tree structure is shown in Figure 3. Note that in the figure, nodes with the same name are stored at the same peer.

4.3. Node Join

A node [2] wanting to join the network must know at least one node inside the network and sends a JOIN request to that node. There are two phases in a new node joining the network. The first phase is to determine where the new node should join. This is done exactly as in BATON except that only routing nodes are used and considered during join process. The cost of this step is $O(logN)$, which is the height of the tree. Details are in Algorithm 1.

After determining a position for the new node, the second phase starts. At first, the data node at the position of the

[2] Due to the fact that each peer node is associated with only one routing node and vice versa, we shall simply refer to "routing node" and "peer node" as "node" when such reference does not cause any confusion. We stick to the term "data node" as is.

Algorithm 1 Join(node n)

```
If (Full(LeftRoutingTable(n)) and
   Full(RightRoutingTable(n)) and
   ((LeftChild(n)==null) or (RightChild(n)==null))
  Accept new node as child of n
Else
  If ((Not Full(LeftRoutingTable(n))) or
     (Not Full(RightRoutingTable(n))))
    Forward the JOIN request to parent(n)
  Else
    m=SomeNodesNotHavingEnoughChildrenIn
       (LeftRoutingTable(n), RightRoutingTable(n))
    If (there exists such an m)
      Forward the JOIN request to m
    Else
      Forward the JOIN request to one of its
      adjacent nodes
```

new routing node splits its covered region into two sub regions using *node splitting* algorithm. After that, a new routing node is created to replace the data node; two new data nodes are created and linked as children of the new routing node; sub regions and data covered by these regions are assigned to two new data nodes. Finally, the new routing node and its left child are given to the new node. The other data node is given to the correspondence right adjacent routing node. In case the new node is the first child of its parent, the parent node increases its height by one and notifies its parent. That parent in turn checks to see if its own height is increased in consequence, and if yes, notifies its own parent, and so on recursively.

For example, assume that node u wants to join the network and it sends a JOIN request to node b as in Figure 4. b has its routing tables full and its child slots full, and all its peers in its routing tables (only c in this case) also have both child slots full. Therefore b forwards the request to an adjacent node p. (Note that routing and data nodes alternate, so the adjacent node is really the data node p', which is located at the same peer node as the adjacent routing node p). At p, because its routing tables aren't full, it forwards the request to its parent j, which then forwards the request to n. At n, its routing tables are full but its doesn't have any routing node children. (Data nodes are not considered in this algorithm). So it accepts the new node, as its left child. A new routing node u is created and the data currently covered by its corresponding data node n' is split into two sub-regions, one is assigned to u' while the other remains assigned to n'.

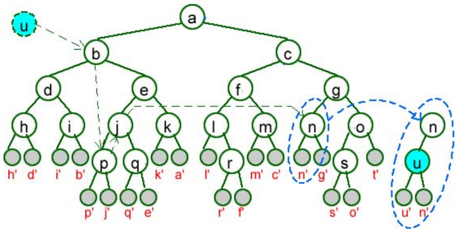

u
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
n
p
q
r
s
u
h' d' i' b' k' a' l' m' c' n' g' t'
p' j' q' e' r' f' s' o' u' n'

Figure 4. A new node joins the network

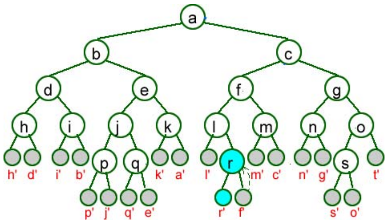


Figure 5. A leaf node leaves the network

The cost of updating routing tables to reflect the existence of the new node is $6logN$ of which $2logN$ is for updating routing tables of the parent node's neighbors, $2logN$ for updating routing tables of the new node's neighbors and $2logN$ for setting routing tables for the new node. For updating height values of sub-trees if their heights change, it takes maximum $logN$ cost if the updating process propagates from a leaf to the root.

4.4. Node Departure

Similar to node join, only routing nodes are used and considered for node departure. If a leaf routing node, which is a routing node without any routing child (or a peer containing a routing leaf node), wishes to leave the network, and there is no routing node in its routing tables having routing children, it can leave the network without affecting the tree balance. In this case, the data node stored at the departing peer will be merged with its data node sibling. The result data node will replace the position of the routing node. For example, assume that a peer containing routing node r wishes to leave the network as in Figure 5. It's clear that node r can leave without affecting the tree balance (note that only routing nodes are considered here). Thus, the corresponding data node r' is merged with its data node sibling f', which later is pulled to replace the position of r. The cost of this process is $4logN$ in which $2logN$ is for updating routing tables of departure routing node's neighbors and $2logN$ for updating routing tables of neighbors of the departure routing node's parent. In addition, as in the case of a node join, a maximum additional $logN$ cost may be incurred for updating height of sub-trees if there are any changes due to the node departure.

If a routing node wishing to leave is an internal routing node or a leaf routing node whose neighbor nodes have routing node children, it needs to find a replacement that is a routing leaf node. Algorithm 2 shows how the process of finding replacement node incurs $O(logN)$ cost. The cost of updating routing tables to reflect changes is $8logN$ of which $4logN$ is for updating routing tables of neighbors of the replacement node and its parent, and $4logN$ for updating routing tables of neighbors of the departure node and its parent. Note that upside paths don't need to be updated in this case since upside paths only keep covered regions of ancestors, not physical references to them.

Algorithm 2 FindReplacementNode(node n)

```
If (LeftChild(n)!=null)
  Forward the request to LeftChild(n)
Else If (RightChild(n)!=null)
  Forward the request to RightChild(n)
Else
  m=SomeNodesHavingChildrenIn
      (LeftRoutingTable(n), RightRoutingTable(n))
  If (there exists such an m)
    Forward the request to a child of m
  Else
    Come to replace the leave node
```

4.5. Node Failure and Fault Tolerance

Failure recovery in VBI-Tree is identical to that in BATON. When a node is found to be unreachable, a report is sent to its parent, which is in charge of the recovery process. The parent of the failed node finds a replacement node if necessary and can re-establish links through redundant information at the nodes in its routing tables, and their children. Operations executed during the recovery process may by-pass failed nodes via either the sideways axis by links in routing tables or the up-down axis by parent-child and adjacent links.

5. Index Construction

Given the overlay network described in the preceding section, we describe here how to use it to construct a generic (multi-dimensional) index. The basic idea is to assign a region of the attribute space to each data node. Each internal node has associated a region that covers all regions managed by its children. Many known index structures follow this sort of region assignment – the difference is in the specifics of how these regions are chosen and split – we allow the same set of choices with regard to the specifics. In this manner, we can create a VBI-R-Tree, a VBI-M-Tree, a VBI-SS-Tree, etc. Initially, the root is the only data node and it covers the entire domain. When new nodes join, the domain is split into smaller regions as discussed in Section 4.3. When peer nodes leave, regions are consolidated as described above in Section 4.4.

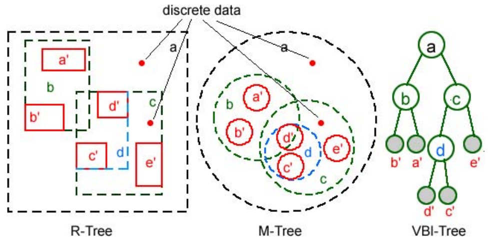


Figure 6. Two dimensional index structure

Addition (or deletion) of data items is performed as in the corresponding centralized indexing scheme. However, there is one issue: when a newly inserted data item at a node has a value that doesn't fall into any region covered by a node's children, one of the node's children should be selected to enlarge its covered region. In a distributed system, such an enlargement can be expensive, involving an update to all upside paths of that node's descendants. To avoid this problem, we propose the use of *discrete data*. Discrete data is defined as data that does not fall into any regions covered by the node's children and hence can be stored at a non-leaf node [3]. A routing node can keep discrete data, and only if the number of discrete data items kept at a routing node exceeds some threshold is a batch enlargement performed of the children's regions. Additionally, we also perform lazy updating. When the region covered by a node is enlarged, update messages aren't sent immediately if the network is busy: rather they are sent to descendant nodes later when the network is free. This laziness does not save in the total number of messages, but simply permits quicker response to a data insertion that causes discrete data to go beyond threshold, and avoids stressing the network at times of load.

The general index scheme is illustrated in Figure 6. In this figure, we show the way to map an R-Tree and an M-Tree in two dimensional space into our framework. We have chosen the data such in the two cases that the resulting VBI-Tree is the same for both schemes. There are two discrete data objects in the figure: one is stored at the routing node a, the other is stored at routing node c.

5.1. General Point Query Process

For simplicity, we first consider the case where sibling nodes do not have overlapping regions, to develop Algorithm 3. For a point query issued or received at node n, if

[3] This is usually not a good idea in a centralized database since internal nodes are usually kept in the memory so that size is a constraint, and high fan-out a more important imperative. But these desiderata do not apply in P2P systems

the region associated with n covers the point query, then the search point will be in the tree rooted at n, and hence the request is either processed at n itself or forwarded to one of its children. If the region associated with n does not cover the point query, then n needs to find the nearest ancestor x covering the point query first by consulting its upside table. After that, it forwards the request to a neighbor node y, found in n's sideways routing table, situated at the other side of the tree rooted at x. Now y uses exactly the same algorithm as n, and continues the search. In order to avoid receiving back the search message from y when searched data is discrete data stored at the common ancestor x of nodes n and y, we use the *nearest-checked-ancestor* parameter, which indicates the parent of the root of the tree to which the search is to be limited. Initially, this parameter is set to null.

Algorithm 3 PointQuery(node n, point p, nearest-checked-ancestor a) //without overlapped regions

```
If (Region(n) covers p)
    If (Region(LeftChild(n) covers p)
        PointQuery(LeftChild(n), p, LeftChild(n))
    Else If (Region(RightChild(n) covers p)
        PointQuery(RightChild(n), p, RightChild(n))
    Else
        LocalSearch(n, p)
Else
    For i = 0 to Level(n) - Level(a) - 1
      x = Upsidepath(n).get(i)
      If (Region(x) covers p)
          If (x == a)
              LocalSearch(x, p) //finding discrete data
          Else
              y = get a neighbor node in other side
                  of the tree rooted at x
              If (y!=null)
                  PointQuery(y, p, x)
              Else
                  PointQuery(Parent(n), p, x)
          Break
```

We illustrate the search process using Figure 7. Suppose node h wants to search a point stored under region covered by node g in an R-Tree index scheme for two dimensional data. Since the search point isn't covered by h's region, h checks its upside path and discovers that the only region that covers the search point corresponds to node a. As a result, h forwards the point query to j, which is its neighbor node on the other side of the subtree rooted at a. Now j checks its upside path and realizes that the nearest ancestor node whose region covers the search point is c. However, it cannot find a neighbor routing node on the other side of the tree rooted at c. Thus, it forwards the query to its parent f. Finally, f forwards the query to g, which is the destination routing node, and g forwards the request to the data node l'.

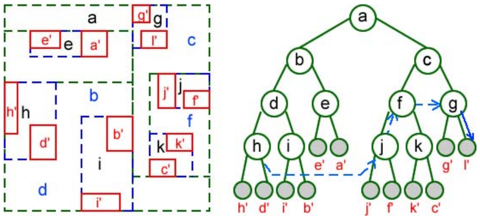


Figure 7. Point query search

With the above algorithm, when a node n wants to search for a point, if the search point is covered by the region of n the cost of search process is $h \leq logN$, where h is the height of the subtree rooted at n. If the search point is neither covered by the region of n nor by the region of the root, by checking the upside path and the routing tables, it takes only one step to forward the request sideways to a node under control of a subtree whose height is less than the height of the current search tree by at least one (limited by the *nearest-checked-ancestor* parameter). In the last case, if the search point is covered by the region of the root, it takes one step to forward the request to an n's neighbor node. After that, the request is forwarded upward to the root, which take $logN$ steps. As a result, the total cost of search process is $O(logN)$. Note that the process of forwarding the search request from a node to its ancestor can be reduced by taking adjacent links among nodes instead of using only child-parent links.

A worry with a tree-structured overlay network is that the small number of nodes near the root will have to do disproportionately more work. However, such a case doesn't happen in our algorithm since search requests are forwarded up only if the recipient node itself has discrete data relevant to the query, and so has to be consulted; or the current node has no correspondence neighbor node to forward the request so that what would have been a sideways forwarding has to be kicked up a level. In the latter case, the node has to be a leaf node, which far enough away from the root.

So far, we have discussed the point query search algorithm without overlapping regions. However, most multi-dimensional indexing schemes allow regions associated with index nodes to overlap. Due to this, queries may be forwarded to multiple nodes instead of only one node. We use the same mechanism in our framework. Because the point query search can be considered as a special case of range query search in which the range query radius is 0, we skip a walkthrough of this algorithm, and move on to range queries.

5.2. General Range Query Process

This section gives a general algorithm for range query processing. The search range algorithm is described in Algorithm 4. Obviously, one now seeks nodes with regions that intersect the query region rather than contain the query point. Since many nodes may intersect with a given query region, the request may be forwarded to multiple nodes. For example, assume that in an R-Tree index scheme for two dimensional data, node h wants to search a region as in Figure 8. At first, h executes the query locally as its covered region intersects with the searched region. After that, it tries to forward the query to other nodes. Since d' is a child of h and its region intersects with the searched region, d' is forwarded the query. Because all of h's ancestors intersect with the searched region, h forwards the query to i, j, which are neighbor nodes located on the other side of the subtree rooted at d and a. h also forwards the request to its parent d because it can not find a neighbor node located on the other side of the subtree rooted at b. Thereafter, the request is forwarded to e from d, to g from j through f. Finally, at destination nodes i, e, and j, the query is forwarded to both data nodes b', r' and l', and routing nodes b, c for discrete data.

Algorithm 4 RangeQuery(node n, search-region r, nearest-checked-ancestor a)

```
If (Region(n) ∩ r != ø)
    LocalSearch(n, r)
    If ((Region(LeftChild(n)) ∩ r != ø)
        and (The search request is not sent from LeftChild(n))
        RangeQuery(LeftChild(n), r, LeftChild(n))
    If ((Region(RightChild(n)) ∩ r != ø)
        and (The search request is not sent from RightChild(n))
        RangeQuery(RightChild(n), r, RightChild(n))
For i = 0 to Level(n) - Level(a) - 1
    x = UpsidePath(n).get(i)
    If (Region(x) ∩ r != ø)
        If (x == a)
            If (No Region(Children(x) covers the whole r)
                LocalSearch(x, r)
        Else
            y = get a neighbor node in other side
            of the tree rooted at x
            If (y!=null)
                RangeQuery(y, r, x)
            Else
                RangeQuery(Parent(n), r, x)
```

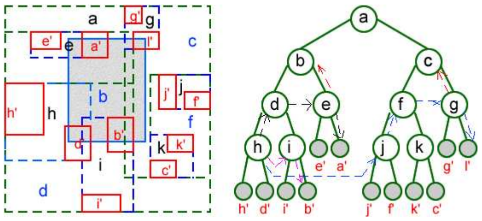


Figure 8. Range query search

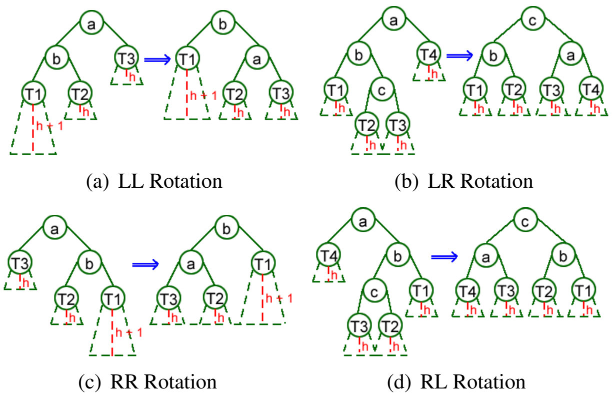
Consider a node x that issues a range query r and for each node y whose region intersects with r. If y is a descendant of x, the search request is always forwarded downward to reach y. If y is an ancestor of x, at first the request is forwarded to a neighbor node z of x, which is on the other side of the tree rooted at y. After that, z checks and realizes that no children of y covers the whole r. Thus, z forwards the request to y via child-parent links. In the last case, if y is neither descendant nor ancestor of x, y must fall into a subtree rooted at z, which is an ancestor of x and z's region must intersect with r. As a result, the request is forwarded to a neighbor node t, which is on the other side of the tree rooted at z. This process continues until y is either ancestor or descendant of the node receiving the searched request. In consequence, the algorithm is guaranteed to forward the query request to all nodes intersecting with the searched region. Even though this algorithm is a little different from the point query search algorithm without overlapped regions because at each step a search request may be forwarded to multiple nodes, the cost of the search algorithm is still kept at $O(logN)$ hops since these requests are forwarded in parallel.

6. Load Balancing

There are two load balancing schemes in VBI-Tree. The first scheme is simple, the overloaded node tries to do load balancing with its children if it is an internal routing node, or with its sibling if it is a data node. Since this scheme is not sufficient to deal with a very skewed data set, we propose the second scheme. In the second scheme, an internal routing node always does load balancing only with its children. However, if it is a leaf data node, it tries to do load balancing with its sibling first. If its sibling is also overloaded, then it finds a lightly loaded data node via neighbor links as in BATON. The lightly loaded node leaves its current position and rejoins as a child of the overloaded node to share the work load. This scheme can quickly redress global imbalances in load, but it can cause the binary tree to become unbalanced in depth: network restructuring is thereafter used to rebalance the tree.

6.1. Network Restructuring

When a node receives a message from its child to update its height, by comparison with the height of the other child,



(a) LL Rotation (b) LR Rotation

(c) RR Rotation (d) RL Rotation

Figure 9. Network Restructuring

a node can know if its subtree is still balanced. If the system becomes unbalanced, network restructuring is started, by the node that detects this, using rotation operators as in the AVL tree [11]. There are four ways to rebalance the tree as shown in Figure 9.

In the LL Rotation and RR Rotation, two nodes are required to recalculate their covered regions. As in Figure 9a and 9c, a needs to recalculate the minimum region covering $T2$ and $T3$ first. After that, b recalculates the minimum region covering $T1$ and a. No data movement is required except for discrete data locally stored at a and b. Similarly, in the LR Rotation and RL Rotation, shown in figure 9b and 9d, first b and a recalculate the minimum region covering $T1$ and $T2$, $T3$ and $T4$. Then, c recalculates the minimum region covering b and a. Finally, discrete data stored at a, b, and c is appropriately reallocated. After getting new covered regions, these regions need to be updated in the upside tables at descendant nodes. As in the case of enlarging covered regions due to data insertion, lazy updating can be used. For a network restructuring involving n nodes, n messages are required to do updating. Additionally, each node also needs to notify their new neighbor nodes about changes which takes $O(logN)$ cost. As a result, a network restructuring involving n nodes requires $O(n \times logN + n)$ effort. The more the nodes participating in the restructuring process, the more the cost of restructuring. However, the probability of the rotation process involving n nodes exponentially decreases with n. The amortized cost of an update can be shown to be just $O(logN)$.

7. Experimental Study

We built a peer-to-peer simulator to evaluate the performance of our proposed system over large-scale networks. For each experiment in a multi-dimensional space, 100000 data objects are inserted into a network of 10000 nodes. Against this system, 1000 point queries, 1000 range queries, and 1000 kNN queries are executed. Using the VBI framework, we implemented the M-tree [7]. A well known P2P system supporting multi-dimensional data - CAN [14] - is used for comparison.

7.1. Performance of Point Queries

Figure 10(a) shows the average and maximum number of hops required to find the result for point queries in different dimensions. The result shows that the VBI-Tree performs independent of dimensionality. CAN achieves good performance only for a large number of dimensions. This may appear counter-intuitive at first, but it is due to the number of neighbors for each node in CAN going up with the number of dimensions, along with an increase in the size of routing tables. Figure 10(b) shows the increase in average number of search hops with increasing network size. CAN is executed in 5, 10, and 20 dimensional space.

7.2. Performance of Range Queries

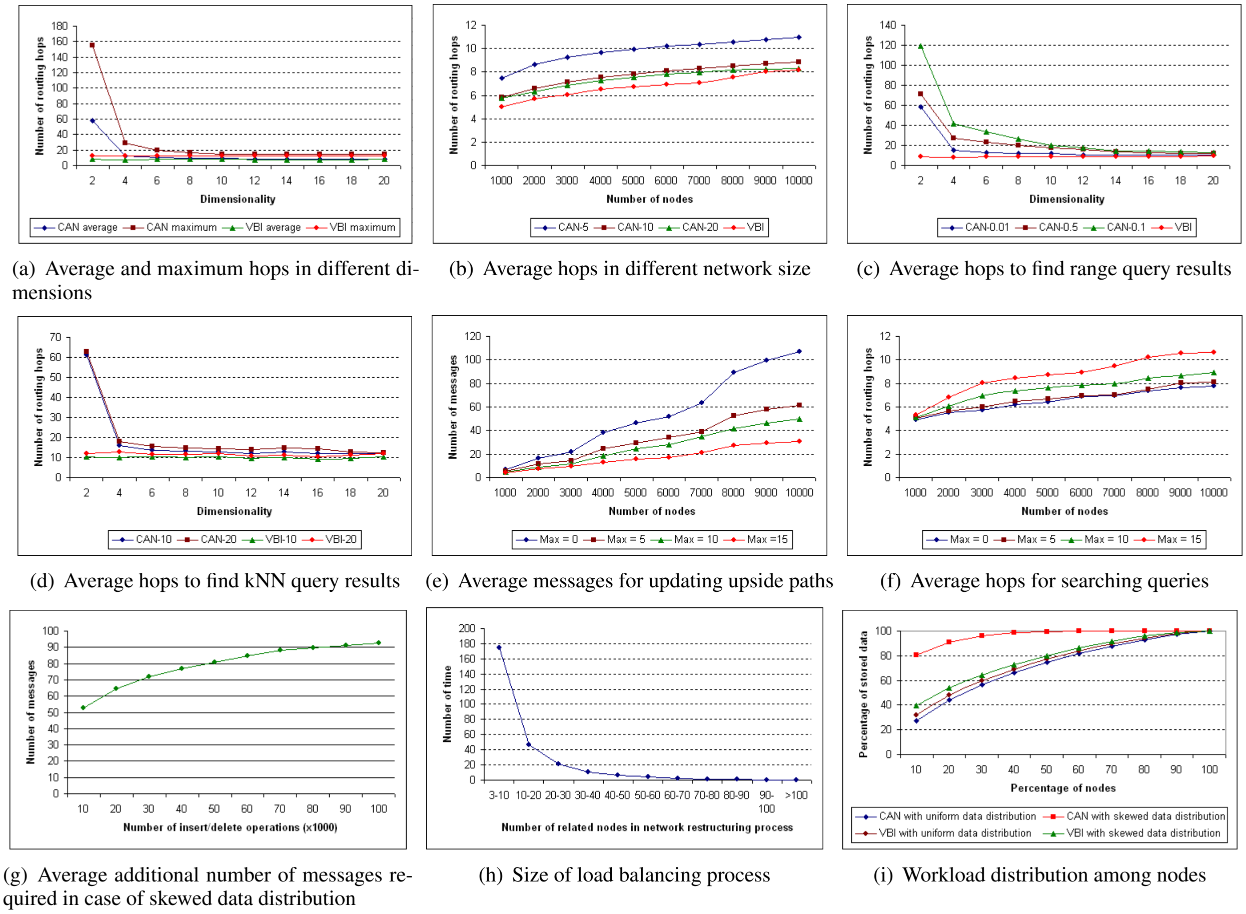
Figure 10(c) shows the average number of hops required to find the result for range queries in different dimensions with different radius sizes. As in the case of point query processing, the result in VBI-Tree is only affected by the size of the network. It is not affected by change of dimension or radius size. In contrast, CAN requires more hops for processing when the size of range queries is bigger.

7.3. Performance of kNN Queries

In this experiment, we implemented a simple kNN query algorithm. In this algorithm all points that fall within a predefined radius are returned. If the total number of returned points is still less than the number specified, the search distance is increased incrementally until enough points are returned. We set the initial radius small enough so that too many points are never returned at first. The result are shown in Figure 10(d) shows that both CAN and VBI-tree increase average number of hops slightly when the number of requested nearest neighbor nodes is increased from 10 to 20. Other characteristics are similar to previous experiments.

7.4. Cost of updating upside path versus Cost of search

Discrete data are an innovation in VBI Tree, and were introduced to reduce the cost of updating upside path tables. However, discrete data increases the cost of search queries. In this experiment, we vary the limit on the size of discrete data allowed at each internal (routing) node. Figures 10(e) and 10(f) show that when we increase this value, the average number of messages required for updating upside paths



(a) Average and maximum hops in different dimensions

(b) Average hops in different network size

(c) Average hops to find range query results

(d) Average hops to find kNN query results

(e) Average messages for updating upside paths

(f) Average hops for searching queries

(g) Average additional number of messages required in case of skewed data distribution

(h) Size of load balancing process

(i) Workload distribution among nodes

Figure 10. Cost of point query search (a, b), range query search (c), kNN query search (d); The effect of varying the number of maximum discrete data (e, f); The effect of load balancing (g, h)

decreases but the number for search increases. Depending on the system, we should adjust the size of discrete data.

7.5. Effect of load balancing

Figure 10(g) shows the average additional number of messages required to do load balancing and network restructuring in case of skewed data distribution compared to uniform data distribution. The result shows that the additional cost required for doing load balancing and network restructuring is not much, around 1 message for every 1000 insertion/deletions. This cost is so low because most of the time, network restructuring only involves a small number of nodes as shown in Figure 10(h).

7.6. Workload distribution

To evaluate the workload in case of skewed data distribution, we tested the network with two sets of data: one uniform and the other skewed. The skewed data set is generated using a Zipfian distribution with parameter 1.0. Figure 10(i) shows that the distribution of data stored at nodes is not very sensitive to data skew in the case of VBI-Tree whereas CAN is highly sensitive to such skew. With the Zipfian data set in CAN, approximately 80% of data is stored at only 10% of nodes. (Note that all preceding experiments were run with uniform data sets and so represent a best case for CAN. Our relative performance would be even better if we used real data with skew). This experiment explains why the original CAN uses DHTs for data distribution, which makes it only suitable for point queries .

7.7. Access load

Figure 11 shows access load of nodes at different levels, measured in terms of the average number of messages received at a node in each level. Level 0 is the root, high num-

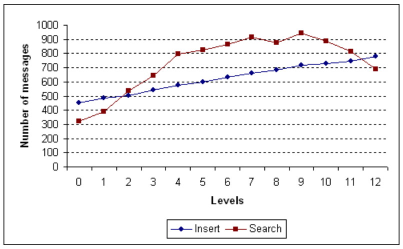


Figure 11. Access load for nodes at levels

bered levels are leaves. In case of insert operation, the result shows that higher level (closer to leaf) nodes always have higher work load. This is due to updates to the upside tables: nodes closer to the leaf have more entries in these tables and so are more likely to have to update these. In the case of search, the load is more evenly distributed, with slightly higher load at levels just above the leaves. In no case is there a bottleneck at nodes close to the root.

8. Conclusion

We have described VBI-Tree, a framework based on a binary balanced tree structure, which can support both point queries and range queries over high dimensional space efficiently. This framework can be used to implement a variety of hierarchical region-based index structures, including M-tree, R-tree, R*-tree, X-tree, etc., in a peer-to-peer system. Experimental evidence supports the effectiveness of this framework. Our contributions include:

- A virtual binary overlay network that is a significant modification of BATON.
- Introduction of discrete data as a mean to minimize update costs and novel P2P search algorithms that account properly for such discrete data.
- An AVL-tree like rotation scheme for rebalancing the virtual binary tree when needed, leading to effective load balance even with highly skewed data.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Proceeding of The 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.

[2] N. Beckmann, H.-P.Kriegel, R.Schneider, and B.Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD*, pages 322–331, 1990.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sep 1975.

[4] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd VLDB Conference*, 1996.

[5] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Cantania. *Indexing Techniques for Advanced Database Applications*. Kluwer Academics, 1997.

[6] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. 33(3):322–373, Sep 2001.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB Conference*, pages 426–435, 1997.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD*, pages 47–57, 1984.

[9] K. Hinrichs and J. Nievergelt. The grid file: A data structure designed to support proximity queries on spatial objects. In *Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*, 1983.

[10] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st VLDB Conference*, 2005.

[11] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Professional, 1998.

[12] J. Lee, H. Lee, S. Kang, S. Choe, and J. Song. CISS: An efficient object clustering framework for DHT-based peer-to-peer applications. In *DBISP2P*, pages 215–229, 2004.

[13] W. S. Ng, B. C. Ooi, and K.-L. Tan. Bestpeer: A self-configurable peer-to-peer system. In *Proceedings of the 18th ICDE Conference*, 2002.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM Annual Conference of the Special Interest Group on Data Communication*, pages 161–172, 2001.

[15] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *Proceedings of the 20th ICDE Conference*, 2004.

[16] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of HPDC-12*, 2003.

[17] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB Conference*, pages 507–518, 1987.

[18] Y. Shu, K.-L. Tan, and A. Zhou. Adapting the content native space for load balanced. In *DBISP2P*, pages 122–135, 2004.

[19] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings. of the ACM SIGCOMM*, 2003.

[20] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th ICDE Conference,*, pages 516–523, 1996.

[21] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical report, Princeton University, 2004.