Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 25 (2009) 77–88
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

An efficient peer-to-peer indexing tree structure for multidimensional data

Rong Zhang[a], Weining Qian[b], Aoying Zhou[b,*], Minqi Zhou[a]

[a] *Department of Computer Science and Engineering, Fudan University, China*
[b] *Software Engineering Institute, East China Normal University, China*

Received 31 August 2007; received in revised form 7 February 2008; accepted 16 February 2008
Available online 10 March 2008

Abstract

As one of the most important technologies for implementing large-scale distributed systems, peer-to-peer (P2P) computing has attracted much attention in both research and industrial communities, for its advantages such as high availability, high performance, and high flexibility to the dynamics of networks. However, multidimensional data indexing remains as a big challenge to P2P computing, because of the inefficiency in search and network maintenance caused by the complicated existing index structures, which greatly limits the scalability of applications and dimensionality of the data to be indexed.

We propose *SDI* (Swift tree structure for multidimensional Data Indexing), a swift index scheme with a simple tree structure for multidimensional data indexing in large-scale distributed systems. While keeping the query efficiency in $O(\log N)$ in terms of routing hops, SDI has extremely low maintenance costs which is proved through theoretical analysis. Furthermore, SDI overcomes the root-bottleneck problem existing in most other tree-based distributed indexing systems. Extensive empirical study verifies the superiority of SDI in both query and maintenance performance.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Multidimensional data; Peer-to-peer (P2P); Point query; Range query; Distributed networks

1. Introduction

The improvements in broad-band network connectivity and the storage capability of computers have resulted in increasing demands on data management in large-scale distributed systems. Because of the massive scale of distributed data, it can easily overwhelm the storage and processing capability of any single node. Peer-to-Peer (P2P) systems open an exciting possibility for such data management tasks, due to the advantages they provide, such as high availability, high performance achieved from large-scale parallel processing, and high flexibility to the dynamics of networks.

Though current P2P systems have achieved great success in file-sharing and file management with the help of mature technologies such as keyword-based search and one dimensional data indexing, extending P2P technologies to applications with more complicated data management tasks is nontrivial. There are several difficulties in implementing multidimensional data indexing and supporting multidimensional complex/similarity queries, which can be either k-nearest-neighbor (KNN) queries or range queries. More formally, a multidimensional space is a pair $M = (D, d)$, where D is the domain objects and d is the distance function. Let $\Phi \subseteq D$ be a subset of D indexed by an index structure.

Definition 1 (*Range Query*). $R(q, r)$ $q(q \in D)$ retrieves all elements that within distance r: $S = \{p \in \Phi | d(q, p) \leq r\}$.

Definition 2 (*K-Nearest-Neighbor(KNN) Query*). $\mathrm{KNN}(q, k)q$ $(q \in D, k > 0)$ retrieves a data set $KS \subseteq \Phi$: $|KS| = k$, $\forall x \in KS, \forall y \in \Phi \setminus KS, KS = \{x \in \Phi | d(q, x) \leq d(q, y)\}$.

For these queries, we shall get a set of data objects that are the most relevant to the search criteria according to some semantic distance function. Almost every existing overlay network protocol underlying the structured P2P systems employs a one-dimensional identifier (or ID, for short) space. The only exception protocol, CAN [22], uses a low dimensional

* Corresponding author.
E-mail addresses: rongzh@fudan.edu.cn (R. Zhang), wnqian@sei.ecnu.edu.cn (W. Qian), ayzhou@sei.ecnu.edu.cn (A. Zhou), zhouminqi@fudan.edu.cn (M. Zhou).

0167-739X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2008.02.010

torus as the topology of the ID space. The dimensionality of the ID space usually cannot be matched with the dimensionality of the data to be indexed. Apparently, these distributed-hash-table (DHT) based methods cannot be used directly for our purpose. Since hashing may destroy the locality of data, range-based queries cannot be supported.

A natural adaption of DHT-based methods is to replace the hashing with continuous mapping. Even then, the mismatching between the dimensionalities of ID space and data space is still a problem. Though there are dimensionality reduction techniques, such as space-filling curves, they usually destroy the locality of data, and thus result in an inefficiency in search, especially for high-dimensional data.

Another natural choice is to extend the tree-based indexing in centralized databases or traditional distributed databases with a limited number of nodes. Several efforts have been devoted to this approach, such as BATON [8], VBI-tree [9] and P-tree [3]. Search efficiency, load balancing, fault recovery, and root-bottleneck issues are studied. Essentially, these methods imitate a multidimensional tree index with an additional ring-based overlay linking all nodes in the tree together. Efficient search and network maintenance algorithms are invented. However, these methods suffer from the problem of high maintenance cost, due to the maintenance cost for the tree structure, for the underlying ring, and for keeping consistency between these two schemes.

We argue that tree structured overlay network should be used in the way that additional links should be added with care. With appropriately designed algorithms, a simple tree structured overlay network with less links could be more efficient than a complex one in terms of, not only network maintenance, but also multidimensional data search.

In this paper, we present SDI, a Swift tree structure for multidimensional Data Indexing in large-scale distributed systems. Its novel features are listed as follows:

- SDI has a succinct tree structure with few additional network links. With carefully designed routing links, they will not overload low-level nodes such as the root of the tree. Thus, its maintenance overhead is low.
- New query processing algorithms are proposed, which bound the search cost by $O(\log N)$, defined as the maximum path length of finding the answer for the query in a network of N nodes.
- SDI supports not only point queries, but also range queries and KNN (K-nearest-neighbor) queries. It scales well in terms of query radius, network size, data distribution and dimensionality.
- SDI preserves the advantages of tree-based indexing techniques, including load balance, fault tolerance, and search efficiency.

Extensive experiments validate the efficiency and effectiveness of this proposed approach.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 introduces the background knowledge of VBI-Tree. Section 4 provides the SDI structure. Section 5 explains the system building in detail. Section 6 describes query processing in the new architecture. In the end, Section 7 shows experimental study, and Section 8 gives a conclusion.

2. Related work

In P2P systems, a good multidimensional data indexing structure must have the following primary characteristics: scalability, load balancing, efficiency and location sensitivity (for range query). However most of existing methods cannot guarantee all these. Current P2P multidimensional indexing structures are mainly based on the methods used in centralized systems [2,21,10] and traditional distributed domains [7,18,16]. We can divide them into two main categories.

The first one is the dimension-reducing based method. [23,13,4] using Space Filling Curve to transform the multidimensional data into one-dimensional data, [24,25] make use of pivot points to do data transformation and [14] compares four kinds of designs which make use of dimension reducing techniques or pivot points. After that, a classic index will be used to index the data. But their performance will be reduced when facing highly skewed data or requiring data preprocessing in a centralized fashion. [27] proposes a three-level cluster scheme using distributed statistics, and it uses a super peer to improve the search efficiency. MAAN [28] and Mercury [29] use multiple one-dimensional index structures to index multidimensional data. But they have high cost when dimensionality increases. The second one is tree-based method. CAN [22] can be considered as the first system, which supports multidimensional data indexing. It is something like KD-tree [11]. Refs. [5,1,20] are all based on CAN. [1] uses the inverse Hilbert transform to map one dimensional data space to CAN. Ref. [20] is proposed for caching low-dimensional range queries. pSearch [5] is proposed for document retrieval in P2P networks, but it does not focus on range queries or KNN queries. SkipIndex [6], EZSearch [12,15] are KD-tree based methods and node IDs are assigned based on the data distribution. However, if the data distribution changes, these systems must be reorganized. In [26] each peer maintains an instance of an Address Search Tree, which reduces the system scalability. The hierarchy-tree-based approach [19,9,17] organizes the nodes into a hierarchy and provides a node-to-index mapping to this hierarchical structure. Using the hierarchy-based approach is potentially more suitable for multidimensional data management, because traditional multidimensional indexing has been well-documented. However, [19] has not mentioned maintenance costs for the dynamic R-tree, and [9,17] are both based on tree branch to resolve queries, which will incur expensive updating costs when a lower level nodes' state change.[1]

3. Background knowledge—VBI-Tree

First, we present a brief review of the relevant features of VBI-Tree. VBI-Tree, shown in Fig. 1, is a variation of BATON

[1] We define root as on the lowest level and the leaves are on the highest level.

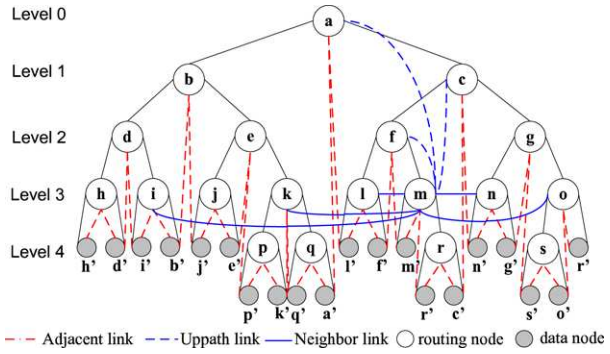


Fig. 1. VBI-Tree structure.

[8], which can be used to implement any kind of hierarchical tree structure that has been designed based on the space containment relationship. There are two main components in the framework. The first one is the overlay network based on the balanced binary tree; the second one defines abstract operations for multidimensional indexing. Each node, except the peer keeping the right most data node, in the network plays two roles: a data node and a corresponding routing node. The parent node of the leaf/data node is called the **leaf routing node** (**LRN**). In Fig. 1, nodes with the same names are maintained by the same peer. If a routing node is not an LRN, it has the same maintainer as the left branch's right most data node, or else it has the same maintainer as its left child data node. The special case is the right most data node which has no corresponding routing node. All requests forwarded to this node will be dealt with by its parent. The data/leaf node is used to store data and the routing node is a virtual node, which only maintains region information called routing space. If it travels upwards, the data will always be covered by the ancestor nodes. The pair *(level, number)* represents a node logically and exclusively. The root is defined on level 0, and the children are one level greater than parent level. Each level has at most 2^L nodes with L as the level number. It numbers each position at Level L from 1 to 2^L, from left to right, whether or not it is vacant currently. For a routing node, it may connect to other nodes by up to four different kinds of real links and one kind of virtual link: **parent link**, **children links**, **adjacent links** which connect the data nodes and routing nodes alternately by an in order traversal, **neighbor links** which is the same level nodes with number less or greater than current node by 2^i $(0 \leq i \leq L)$, and the virtual **upside path links** which just log all the ancestors' region coverage information along its way to root (has no real link). It puts the neighbor nodes which have numbers less or greater than the current node by a power of 2 in left or right routing tables. If there is no such neighbor node, It just puts "null" in the corresponding position. A routing table is full if all valid neighbor positions are not null. In VBI-Tree, it creates a new data structure **discrete data**, which is used to keep the data belonged to no child spaces temporarily, to relax the updating cost. No updating messages are broadcasted to its descendants until the number of discrete data exceeds a certain threshold.

For m in Fig. 1, it has 1 parent link to f, 2 children links to m' and r, 2 adjacent links to m' and r', 3 neighbors $\{l, k, i\}$, which are 2^i distance from m with $i = \{0, 1, 2\}$ in left routing table and 2 neighbors $\{n, o\}$, which are 2^i distance from m with $i = \{0, 1\}$ in its right routing table and 3 virtual upside path links to f, c and a.

There are two important rules in VBI-Tree. The first one is that a tree is balanced if every routing node which has a child has both its left and right routing tables full. The second one is that if a node x contains a link to another node y in either its left or right routing table, the parent node of x must contain a link to a parent of y unless they share the same parent node. The first rule gives a way to process node join or departure so as to guarantee tree balance; the second rule gives an efficient way to do fault recovery and query forwarding.

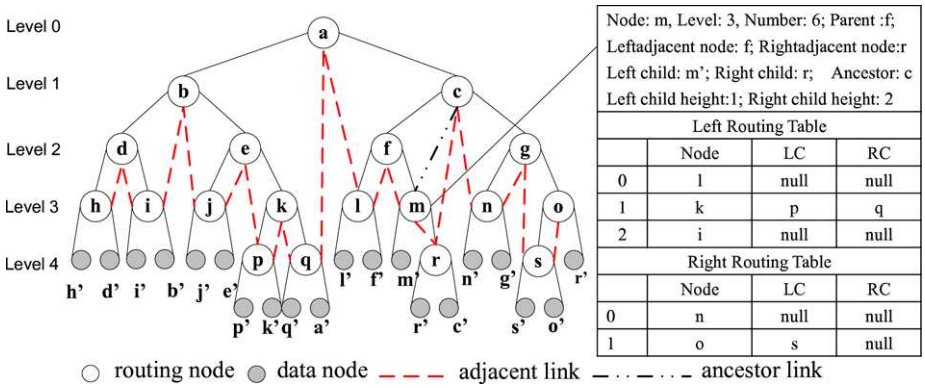
In VBI-Tree, we can notice the following problems:

- Each node keeps an eye on all the ancestors which lie on the way to root, so each expansion or shrinkage to any ancestor will cause all the descendants to update the corresponding upside path to the ancestor. So the updating cost is bounded by $O(N)$.
- There are only virtual links among descendants and ancestors, so discrete data checking message climbs up the tree one level at a time, which costs many redundant messages before it reaches the target node.
- Routing nodes link data nodes using adjacent links, which has mixed the structure and not restricted all the routing to the routing layer.

4. SDI architecture

In order to overcome the shortcomings of VBI-Tree, we put forward SDI, which is shown in Fig. 2. Each routing node may "link" to five kinds of nodes, if any: one parent, two children, two adjacent routing nodes, neighbor routing nodes and one ancestor node. Compared with VBI-Tree, SDI defines new ancestor links and different adjacent links, but removes upside path. If we remove all data nodes which are leaves, we get a **routing tree** with only routing nodes. By an in-order traversal to this routing tree, we create **adjacent link** between any two nodes as shown in Fig. 2. Given a node x, the nodes immediately prior to and after it, connected by the adjacent link, are the left and right adjacent nodes, respectively. Adjacent link will always connect to one *LRN* at one end. **Ancestor link** is distributed by the lower level routing node to its specific higher level routing node descendants, which are at least two levels higher[2] and lie on the left(right) child branch but right(left) most positions at each level. Ancestor link brings the ancestor coverage information to its selected descendants. For any node at level l ($l >= 0$), it will distribute at most $2 * (\log N - l - 2)$ links out with network nodes N. Child height is the child subtree height, which is used to activate the balance algorithm [9]. For the data node, it has no sideways routing tables, ancestor or adjacent links, but only one parent link to *LRN*. Modification to the original adjacent link has restricted the routing inside routing tree, which makes the tree succinct and using ancestor link instead of upside path reduces

[2] The farther the node from the root, the higher the level it has.



Node: m, Level: 3, Number: 6; Parent :f;
Leftadjacent node: f; Rightadjacent node:r
Left child: m'; Right child: r; Ancestor: c
Left child height:1; Right child height: 2

Left Routing Table

	Node	LC	RC
0	l	null	null
1	k	p	q
2	i	null	null

Right Routing Table

	Node	LC	RC
0	n	null	null
1	o	s	null

Fig. 2. SDI Structure.

the updating cost from $O(N)$ to $O(\log N)$, which makes the tree swift.

In Fig. 2, we show m's routing information. It has full left and right routing tables and one ancestor link to c. *LC* and *RC* are the left and right child information for the neighbor, but notice that only routing node is effective for these two items. l, the first neighbor in the left routing table, has only data node children. So we set "null" to its *LC* and *RC*, respectively.

4.1. Ancestor link distribution

Only routing nodes are considered here. Supposing (l_r, n_r), (l_a, n_a)[3] denote the node and the ancestor it links to, respectively, with $l_r > l_a + 1$ and $l_a >= 0$, we define the corresponding relationship as:

$$n_r = \begin{cases} (n_a - 1) * 2^{l_r - l_a} + 2^{l_r - l_a - 1}, & n_r \text{ is even} \\ (n_a - 1) * 2^{l_r - l_a} + 2^{l_r - l_a - 1} + 1, & n_r \text{ is odd.} \end{cases} \quad (1)$$

For a given node (l, n), by Eq. (1) it is easy to detect whether it has any ancestor link and if any, what level the ancestor locates (Theorem 1). Such ancestor links are helpful to reduce the low level nodes' load (Theorem 2).

Theorem 1. 1. *All nodes with level no less than* 2*, except the left most and right most ones, will have ancestor links.*
2. *If one node maintains an ancestor link, there is only one.*

Proof. Supposing (l_1, n_1) has an ancestor link to (l_2, n_2), $(l_1 > l_2 + 1$ *and* $l_2 >= 0)$.
(1) As we have defined, the ancestor node distributes the links to the right/left most node of the left/right child branch at each level. As a left most node, it can never be the left most child of any right child branch at each level. As a right most node, it can never be the right most child of any left child branch. So both of them can not have ancestor links.

For all the other nodes, it can either be the right most child of a left branch or a left most child of a right branch. So they can have ancestor links. □
(2) Supposing (l_1, n_1) maintains the other ancestor link to (l_3, n_3). With Eq. (1), we get

$$(n_2 - 1) * 2^{l_1 - l_2} + 2^{l_1 - l_2 - 1} = (n_3 - 1) * 2^{l_1 - l_3} + 2^{l_1 - l_3 - 1}$$

[3] Node is represented by (level, number) pair logically.

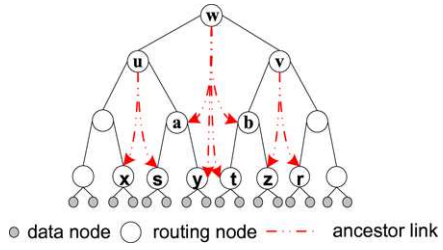


Fig. 3. Ancestor link distribution.

then we get:

$$2 * n_2 - 1 = (2 * n_3 - 1) * 2^{l_2 - l_3} \text{ and } (l_2 >= 0, l_3 >= 0).$$

For both $2 * n_2 - 1$ and $2 * n_3 - 1$ are odd, (l_2, n_2) is the same node with (l_3, n_3). □

In SDI, ancestor links are distributed to routing nodes at each level evenly except the left and right most ones. Each link will be maintained by at most two nodes of the same level: the right(left) most descendant of the left(right) child branch. Let us take Fig. 3 node w for example, it distributes its links to nodes $\{a, y\}$, which are the right most children of the left branch of w at level $\{2, 3\}$ and $\{b, t\}$, which are the left most children of the right branch of w at level $\{2, 3\}$.

Theorem 2. *Supposing there are nodes* x, y *and* z *at level* l_1*, evenly numbered (node* s, t *and* r *oddly numbered) and sibling nodes* u *and* v *at level* l_2 $(l_1 > l_2 + 1, l_2 > 0)$*. Node* w *is the parent of node* u *and* v*.* x, y *and* z *(*s, t *and* r*) are descendants of* u, v *and* w*. If* $x(s)$ *has full routing tables and it has an ancestor link to* u*.*

1. *There must be a neighbor node* $y(t)$*, which has an ancestor link to node* w*.*
2. *There must be a neighbor node* $z(r)$*, which has an ancestor link to node* v*.*

Proof. $N(n)$ denotes the number for node n. Supposing x, y and z have links to u, w and v, respectively:
(1) $N(x) = (N(u) - 1) * 2^{l_1 - l_2} + 2^{l_1 - l_2 - 1}$;
$N(y) = (N(w) - 1) * 2^{l_1 - (l_2 - 1)} + 2^{l_1 - l_2}$;
supposing $N(u) = n_1$ and $N(w) = n_2 \Rightarrow$
$|N(y) - N(x)| = |2^{l_1 - l_2 - 1} * (4 * n_2 - 2 * n_1 - 1)|$;
if n_1 is even, $n_2 = 1/2 * n_1$;

otherwise $n_2 = 1/2 * (n_1 + 1)$.
so $|N(y) - N(x)| = 2^{l_1 - l_2 - 1}$;
for x and y are of the same level, then x and y are neighbors.
So it does to node s, t and r. □
(2) $N(z) = (N(v) - 1) * 2^{l_1 - l_2} + 2^{l_1 - l_2 - 1}$;
supposing $N(v) = n_3 \Rightarrow$
$|N(z) - N(x) = (n_1 - n_3) * 2^{l_1 - l_2}|$;
for node u, v are siblings, $|n_1 - n_3| = 1 \Rightarrow$
$|N(z) - N(x) = 2^{l_1 - l_2}|$;
for x and z are of the same level, then node x and z are neighbors.
So it does to node s, t and r. □

Each node will keep the region information for the linked ancestor, if any. As shown in Fig. 3, x knows the coverage region of u. If x detects the branch rooted at node u can not cover the query and y in x's sideways routing tables, only one hop is needed to neighbor y, which knows the lower level ancestor w's region information. So valid nodes checking to the query will not always be forwarded upwards, which relaxes the load of lower level nodes.

5. Overlay network building

5.1. Node join and node departure

Algorithm 5.1: JOIN (*Node n*, *Contact p*)
if p has full routing tables and has one child routing node vacant **then**
  p accepts n as its child;
else
  if p's routing tables are not full **then**
    Join(n, p.Parent);
  else
    if there is a neighbor node m with no full routing child node **then**
      Join(n, m);
    else
      Join(n, p.higher_level_adjacentnode);

Node join or departure process is similar to VBI-Tree except the routing information updating because of routing links' modification. Currently "upside path" is replaced by "ancestor" item and data node is not linked by an adjacent link any more.

Supposing peer n contacts p, a two-phase joining process will begin. Only routing nodes are considered in this phase. The first phase is to join into the proper position in the tree which will not cause the tree to become imbalanced, as shown in Algorithm 5.1. If it finds a node which has full routing tables but with at least one child routing node vacant, it accepts n as its child. Otherwise, it forwards the request to the parent node, its higher level adjacent node or a neighbor node. The second phase is to reallocate the data space, and to do related routing table and ancestor links updating. p which accepts the joining of n splits its data space, controlled by its data node, into two smaller ones. A new routing node is created to replace p's data node. Now the new routing node and the left child data node are assigned to peer n and the right child of the new routing node is assigned to peer p.

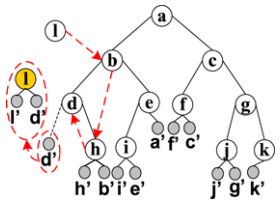


Fig. 4. Node join.

The new routing node will set both its left and right child heights to l and increase its own height by one. In turn, it notifies its parent to check the height. The checking request will not stop forwarding upwards until the height of subtree/tree branch has no change. If the new routing node is oddly numbered, but not the left most one, the ancestor link is the parent's original left adjacent link before updating; if the new routing node is evenly numbered, but not the right most one, the ancestor link is the parent's original right adjacent link before updating. When joining, updating of the ancestor link can be piggybacked over other updating messages, no additional cost is required. The cost reflecting the joining is at most $(6 \log N + 1)$ (N is the network size) messages: among these $2 \log N$ messages are used for updating the parent routing tables, $2 \log N$ messages are used for updating the neighbors routing tables, $2 \log N$ messages are used for building its own routing tables and 1 message is used for getting an adjacent link for the new routing node.

For example in Fig. 4, l sends "JOIN" request to b. According to Algorithm 5.1, d accepts l as its left child. After joining, for l is the left most child of level 3, it need not maintain any ancestor link.

When node leaves, it will take three phases. Only routing nodes are considered also. First, it checks whether it can leave without affecting the tree balance. Second, data space combination is taken. Third, routing table updating starts.

The first case is that a node x can leave its current position without causing the tree to become imbalanced if it belongs to *LRN*s with no neighbors having child routing nodes. After departure, its data node sibling will combine the data belonging to x and replace the original parent node. The cost reflecting such a kind of departure is $4(\log N)$ messages: routing table updating cost for x's neighbors and x's parent's neighbors are $2 \log N$ and $2 \log N$ messages, respectively. There is no ancestor link updating cost.

The other case is that the departure node has to find a replacement node, when x has a child routing node or its neighbors have child routing nodes. If x has child routing nodes, it forwards the "Find Replacement" requirement to the higher level adjacent node, which is an *LRN*, or else it forwards the requirement to a neighbor node which has child routing node. Then the replacement node, which can leave without destroying the tree balance, will leave its current position and take the place of x.

For example in Fig. 5, supposing e leaves the network, it has to find a replacement node before departure. e has a child routing node i, which is the higher level adjacent node. i has no neighbor routing nodes which have any child routing node.

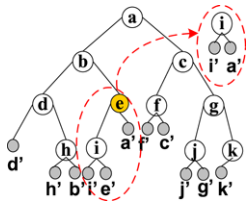


Fig. 5. Node leave.

So i can be the replacement node. After the departure of i, its data node sibling e' combines the with data space of i' and replaces i. Then i comes to replace e both in the routing node position and the data node position. The updating cost for such a departure is $10(\log N) - 4$ messages: $4(\log N)$, $4(\log N)$ and $2 * (\log N - 2)$ to reflect the leaving of the replacement node, the re-joining of the replacement node and the ancestor link updating, respectively.

Either joining or departure costs at most $(\log N)$ messages to update the subtree heights, maybe propagating from the LRNs to root, which is used to detect tree's imbalance.

5.2. Index building

Index building for SDI is the same as in VBI-Tree [9], which deals with the data indexing distribution among tree nodes. The ancestor node will cover the descendant node. Node joining or departure causes the data space to be divided or combined. Data adding or deleting causes the coverage space of node to be expanded or shrunk, which is the same process as in centralized index schemes [2,21].

5.3. Failure recovery, network restructure and load balancing

Besides neighbor links and parent-child links, ancestor links are also used to do failure recovery. We take the similar recovery steps as in VBI-Tree [9]. However, if neither neighbors nor parent can help to do system recovery, it will hop to the ancestor for help. When dealing with load balancing, we can share load between parent and child for internal nodes, and between sibling nodes for data nodes. But for data nodes, we can also do lightly loaded node forceful joining to highly loaded node. When we meet with a tree imbalance during this process, we use the AVL-tree rotation like method to restructure the tree. The details can be found in VBI-Tree [9].

The ancestor links will not be changed much when doing subtree rotation, for the relative node position relations have not been changed much. It means when doing rotation at node n, the nodes which line on the left or right most positions at each branch will not moved to other positions. Only the nodes changing their subtrees after rotation have to update the ancestor links they distribute. Then the updating cost for ancestor links is $O(1)$.

6. Query processing

According to the modification to the tree structure, we define new algorithms for query processing. By visiting ancestor links among neighbors, we promise the query efficiency is still $O(\log N)$ but with no root-bottleneck problem or high maintenance cost. Point query is a special case of range query and KNN query, setting query radius to zero. For simplicity, first we consider the case where no sibling nodes overlap with each other. Later, we will show the general range query algorithm.

There are three different kinds of relationship between two regions: region r_1 and region r_2: **separation**, **intersection** and **coverage**. If r_1 shares no space with r_2, r_1 is separate with r_2; if r_1 shares its whole space with r_2, r_1 is covered by r_2; If r_1 shares part of its space with r_2, r_1 intersects with r_2.

6.1. Point query algorithm

We first present the point query algorithm with no sibling nodes overlapping with each other in Algorithm 6.1. The query forwarding stop conditions are:

- if one of its children can cover the query space completely, we stop forwarding the query to lower level ancestor nodes.
- if sub-root of the checking branch can not intersect with the query space, we stop forwarding the query downwards to higher level nodes.

For a point query issued or received at node n, it will first check its own region. If it covers the query, the query will be either processed by itself or forwarded downwards to its children, if any. Otherwise, it locates the node which can cover it relying on ancestor references.

Algorithm 6.1: POINTQUERY(Node n, QueryPoint q, SubRootPos a, OldAncPos oa, CheckedPath $path$)

```
if n is not in path then
  put n in path;
  if n covers q then
    do subtree search;
  else
    \\ n does not cover q
    if a is the parent of n then
      do sibling and parent checking;
    else
      \\n.lev-a.lev>1
      if n has no ancestor link then
        forward query to sibling or parent;
      else
        anc is n's ancestor;
        if anc covers q then
          get anc's children's maintainers{LCM, RCM};
          if LCM is no in path then
            PointQuery(LCM, q, LC, anc, path);
            anc=anc.rightChild.Pos;
          if RCM is no in path then
            PointQuery(RCM, q, RC, anc, path);
        else
          \\ anc can not cover q
          if anc is a and anc.Lev > oa.Lev then
            nodes[anc.parent to oa] do local search;
          else
            forward query to anc's parent maintainer;
```

Notice that here we identify each tree node by its position, which is the (level, number) pair. There are two cases when n has no reference. The first one happens to the tree/subtree less than three levels high. If its sibling exists and has not been

visited before, we forward the query directly to the sibling and restrict the query to the sibling branch by setting a to n.sibling's position and then n.sibling gets the opportunity to activate the lower level discrete date search to node oa. Or else n starts the ancestor node discrete data search directly. The second one is that it is the left/right most node. We forward the query to its sibling node. If such a sibling node does not exist, the query is forwarded to its parent node. The internal or routing node's local search refers to its discrete data area search.

The next situation is that n can not cover q but it maintains an ancestor link. If n's ancestor *anc* covers q and *anc*'s two children have not been checked, first we do the children checking before jumping directly to *anc* to do discrete data area searching. As we know if any descendant covers q, visiting to its ancestors can be omitted. By setting *oa* to *anc* position, it is easy to turn back to *anc* to do discrete data checking if no children cover q. Any absence of a child maintainer causes the parent node to forward the query. Later we restrict the legal checking branch for the children by setting proper values to parameters *LowestAncPos* and *OldAncPos* so as to avoid redundant checking of messages. If the left child maintainer (*LCM*) has not been visited before, we forward the query to the *LCM* and at the same time we restrict the sub-root position which is under-checking to *anc*'s left child *LC*. The child maintainer with the lower *OldAncPos oa.Lev* value has the responsibility to activate the possible discrete data area checking to the ancestor at *oa* position, then it avoids repeated discrete data searches to ancestors. In the algorithm, we can see that if the left child maintainer has a chance to receive the query, the discrete data area checking to *OldAncPos* will be activated by its left branch descendant. Or else the right branch descendant will get the opportunity. When none of *anc*'s children can cover q, we do discrete data area checking at *anc*.

If the subtree rooted at a has been checked and no coverage relationship exists, it begins discrete data checking to a's ancestors which shall be bounded by *oa*. Notice that all the discrete data area checking stops forwarding upwards once we meet with the nodes which can cover q. The last situation is that when both n and n's ancestor can not cover q, we forward the query to the node which maintains a lower level ancestor reference, if any, until we find the node covering the query.

In order to avoid receiving the search request back, we use parameters *CheckedPath* to log checked nodes, *subRootPos* to mark the checking sub-root and *OldAncPos* to mark lowest ancestor position which maybe needs discrete data area checking. Only if *SubRootPos* sub-tree covers r, nodes listed from *SubRootPos*.Lev - *1* to *OldAncPos* can be deleted from the checking list. The initial values for *CheckedPath*, *SubRootPos* and *OldAncPos* parameters are *null*, 0 and 0, respectively.

Point query example: Let us illustrate the point query algorithm in Fig. 6. This is a 2-dimensional M-Tree [21] based space division. The query is issued at node h, and the target is covered by the discrete data of node a. First, the target is not covered by h. It has no ancestor reference, so it forwards the query to sibling node i, which has an ancestor reference to b. But b can not cover the target. No neighbor of i has lower level ancestor reference. Then the query is forwarded to parent node

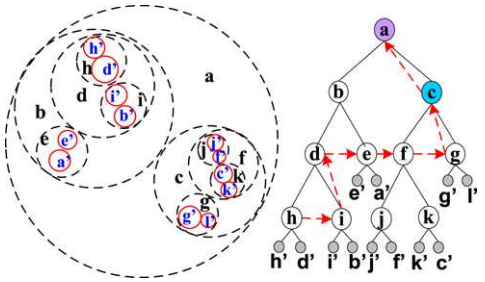


Fig. 6. Point query processing.

d. But d is at the left most position, so d forwards the request of finding a lower level ancestor to node e. Here e finds that ancestor a covers q. According to the algorithm, before doing a's discrete data checking, we shall send the checking request to b's and c's maintainers, which are a's children's maintainers at level two. But neither b nor c has ancestor reference maintained at this level. We choose the left and right direct neighbors, node d and node f, to forward the request. For d has been included in *path*, we only forward the request to f. At f, it has $a = 1$ and $oa = 0$, rooted at node c. There is no ancestor reference distribution at this level. Node g is f's sibling and has not been visited before. Then it receives the search request. After doing a local search, it begins to do an ancestor nodes discrete data area search to node c and node a. At last we get result at a.

In the point query algorithm, the query can be started at any position and the node will only be visited once. If the sibling node or ancestor maintainer is not available at the current level, it will rely on the parent node to forward the query. Ancestor checking need not always be sent upwards, if its corresponding maintainer is found in the sideways routing table. So the root-bottleneck problem is avoided. The search efficiency is $O(\log N)$, where N is the network size.

So far, we have shown the point query processing algorithm without overlapping. We can not avoid overlapping by using general hierarchical space division. In such a case, the query will be forwarded to multiple nodes instead of only one node. We will show the details in range query processing algorithm.

6.2. Range query algorithm

Range query algorithm is shown in Algorithm 6.2. The main difference with Algorithm 6.1 is that even though we find the coverage region, we still can not stop checking and parallel query distribution is taken because of overlapping. Let us say node n issues a range query r.

If n intersects with r, we do local search (discrete data area search) and also begin a downwards subtree search. As we know, if the node intersects with r, all of its ancestors along the way to root will intersect or cover r and then lower level ancestor checking is needed. At this time, we have to send the query to all the other nodes which lie on the other side of the tree rooted at these ancestor nodes to find all results. The query forwarding stop condition has been mentioned in Section 6.1. If n does not cover or intersect with r, we have to rely on the ancestor reference to find the first node which intersects or covers r.

There are two cases that n has no ancestor link under the current searching subtree. The first one happens to the tree/subtree with height less than three and the second one is that it is the left/right most node, which has been detailed in Section 6.1.

Next, node n maintains an ancestor link. If n's ancestor *anc* intersects with r, we do *anc*'s children's maintainers checking, to decide whether *anc* needs discrete data checking. If neither of the one-level higher ancestors can cover r, discrete data area checking is applied to *anc*, which is activated by one of *anc*'s children's maintainers. When finishing a subtree search, rooted at a, discrete data checking starts from *anc.parent* to node *oa*. If *anc* can not cover r, the query is forwarded to *anc*'s parent maintainer to check a bigger region. After we find the ancestor which covers r totally, we distribute the query to the nodes which lie on the other sides of the subtrees rooted at ancestors in *list*, which are still under the tree rooted at a, and at the same time by setting $a = L$ to restrict the search under each sub-tree, it avoids repeated query processing requests. Notice that discrete data area checking to lower level ancestors will stop forwarding once we meet with a node which covers r completely. Additionally, in our algorithms, if the corresponding sibling nodes do not exist, the parent node will take the responsibility of forwarding the query.

Algorithm 6.2: RANGEQUERY(Node n, QueryRegion r, SubRootPos a, OldAncPos *oa*, CheckedPath *path*)

```
if n is not in path then
  if n intersects with r then
    do subtree search;
  if n.Lev is not equal to a.Lev then
    if a is n's parent then
      do sibling and parent checking;
      return;
    if n does not maintain ancestor link then
      forward r to node with ancestor link;
      return;
    if n has an ancestor link to anc then
      if anc intersects with r then
        get anc's children's maintainers {LCM, RCM};
        if LCM is not in path then
          RangeQuery(LCM, r, LC, anc, path);
          anc=anc.righChild.Pos;
        if RCM is not in path then
          RangeQuery(RCM, r, RC, anc, path);
      if anc is a and a does not cover r then
        nodes[a.parent to oa] do local search;
      if anc.Lev>a.Lev then
        if anc does not cover r then
          y=lower Level ancestor maintainer;
          RangeQuery(y, r, a, oa, path);
        else
          {list}=ancestor nodes [anc.Parent to a];
          for each node L in list do
            z is on the other side of the subtree rooted at
            L;
            RangeQuery(z, r, L, L, path);
```

Range query example: Let us illustrate the range query algorithm in Fig. 7. In this example, node h issues a range query r (the shadow ring). At first, h checks itself locally. It neither intersects/covers r nor maintains any ancestor link. h forwards the request to its sibling i, which has an ancestor link to b. However b can not cover or even intersect with r and no other lower level ancestor link maintainer can be found at this level. So it forwards the request to parent d. It is the left most node and does not intersect with r. We forward the request to its sibling e, which maintains ancestor a's region information. it finds a covers the query. After that, we shall check the children of a (b and c). Here b and c are one-level lower nodes to the query issuing node e, and we forward the checking request to the left and right immediate neighbors in e's sideways routing tables. d has been included in *path*. We only forward the request to f. Now the query is restricted to the branch rooted at node c, by setting $a = 1$ and $oa = 0$. f begins the query processing in a shorter tree branch. It intersects with r. So does its sibling g. Neither of them covers r, so we start checking on both g, f and forward discrete data area checking to ancestors c and a. But a receives no discrete data checking request, because c covers r completely. At last, the query is resolved at peers j, f, c and g.

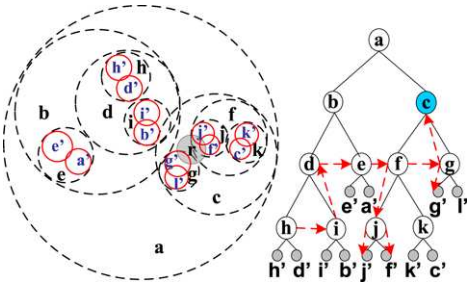


Fig. 7. Range query processing.

6.3. KNN query processing

It is convenient to implement the KNN query by using the range query algorithm.[4] We define that all points falling in a specified radius are returned. If the number of points is not enough, we will increase the radius by a small value δr ($0 < \delta r < 1$) (supposing the data space is a unit hyper space). In order to visit less nodes, we can set the initial radius to a small value.

6.4. Analysis of query algorithms

– Ancestor link plays the most important role in our algorithms. As presented in Section 4.1, ancestor links are distributed evenly to the nodes among the same level descendants and they contain ancestor coverage space information. So checking to a lower level ancestor, whether it intersects or covers the query, can be resolved by the corresponding ancestor link maintainers (higher level descendants) and need not always be forwarded upwards. This solves the root-bottleneck problem. If a level L_1 with no vacant position, this level will contain links to all the

[4] The KNN query can be implemented by more efficient ways. Here we just show a general solution relying on range query processing.

ancestors except to the parent level, that is $2^{L_1-1}-1$. For a node at level L_2, it will distribute at most $2*(\log N - L_2 - 2)$ links to descendants. So SDI's ancestor link updating cost is bounded by $O(\log N)$.

– Ancestor link helps to do a discrete data search with less hops, which reduces the query processing cost drastically, compared with VBI-Tree. For example, in Fig. 2, if m needs do discrete data area checking to a, it hops to c by ancestor link first and then to a. But for VBI-Tree, it hops to f, c and a one by one. The bigger the network size, the more messages are used. And also when doing range query, because of the overlapping, more nodes meet with discrete data checking and then SDI saves more messages than VBI-Tree.
– We guarantee all the nodes which intersect with the query can be visited once and only once. In the query processing algorithm, by using additional parameters, we restrict the search to the nodes or branches which have not been checked.
– For query processing, we use a parallel query distribution which guarantees the query efficiency to be $O(\log N)$.

7. Experimental study

To evaluate the performance of our proposed structure, we implemented a simulation system in Java JDK *1.5*. In our implementation, each peer is identified both physically by a pair of IP address and port number and logically by its position (level, number) in the tree structure. Communication between nodes is via sockets. There is a fake server which distributes events to peer nodes. Peer numbers, discrete data area size, inserted data numbers, dimensionalities and query radius range from 500 to 10,000 (default 2000), from 2 to 20 (defaut 2), from 2000 to 20,000 (default 10,000), from 2 to 20 (default 5) and from 0.01 to 0.2 (default 0.01), respectively. The default point or range or KNN queries are 1000, with $K = 6$ for the KNN query. We generate the inserted data or query data by the Zipfian method with parameter $\alpha = 1.0$ (default), and in some experiments we use $\alpha = 0.4$[5] to produce a set of less skewed data. If there are no special experimental parameters declared, we use the default values. The data space is a unit hyper-sphere. L_2 norm (Euclidean distance) is used to calculate the object distance. We implemented M-tree [21] on both SDI and VBI-tree. And they deal with load balance similarly, for ancestor links in SDI and upside path in VBI have no influence on this aspect. Our comparisons focus on the cost for query processing, data operation and also on load distribution.

7.1. Performance of query processing

VBI-Tree uses upside path to log the coverage information for all ancestors along the way to root and then each node has a wider view. So VBI-Tree beats SDI in query efficiency, as shown in Fig. 8. However both of them can resolve the query by $O(\log N)$. The most important thing for SDI is that

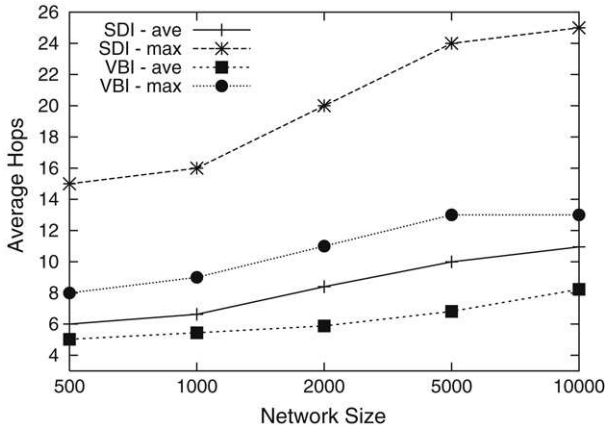


Fig. 8. Average and max. hops with different network size.

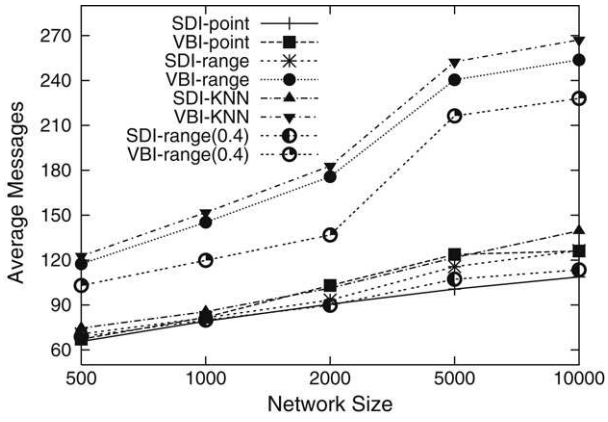


Fig. 9. Average query cost with different network size.

it can resolve the query with much less cost (query messages) and the more skewed the data distribution, the more benefit it gets, as shown in Fig. 9.[6] We implement the KNN query simply with an initial query radius 0.01, $K = 6$ and the radius increase coefficient is $\delta = 0.05$. The bigger the network size, the more dispersive the data and then the more messages are used to resolve KNN query. SDI beats VBI-Tree in three cases, especially for the range query and the KNN query. As stated in Section 6.4, using ancestor links, SDI resolves discrete data checking with less hops but VBI-Tree can only jump one level at a time. When we increase the discrete data size, the average and maximum search hops and the average messages needed to resolve a point query increase, but the average messages needed to resolve a range query do not change much, as shown in Figs. 10 and 11. Fig. 12 shows the range query cost increases as we use a bigger query radius. Fig. 13 shows that the query cost increases with increasing dimensionality, the bigger the dimensionality, the more the space overlapping. KNN query processing adopts the range query processing algorithm and SDI wins in query cost, as shown in Fig. 14.

From the results, we can get that on the one hand VBI-Tree has better query processing efficiency than SDI, but both are bounded by $O(\log N)$; on the other hand, SDI beats VBI-Tree in average messages for resolving point query, range query

[5] We declare it with the figure curve label or else $\alpha = 1$.

[6] In the figure 0.4 is the α value for the Zipfian method.

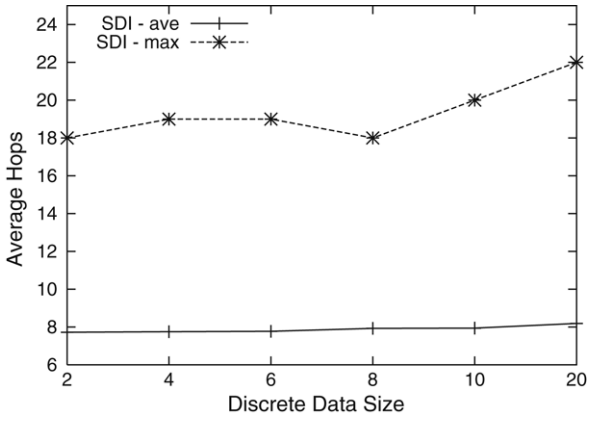


Fig. 10. Average and max. hops with different discrete data size.

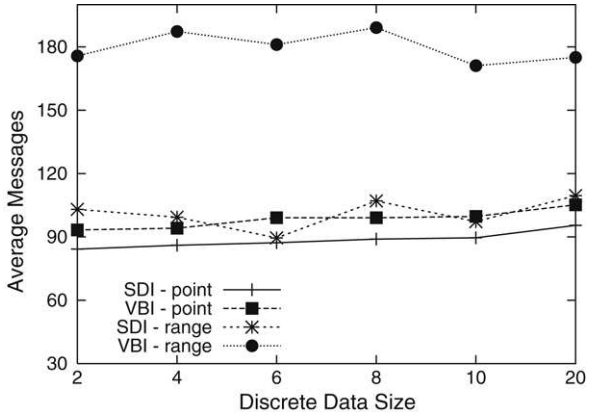


Fig. 11. Average query cost with different discrete data size.

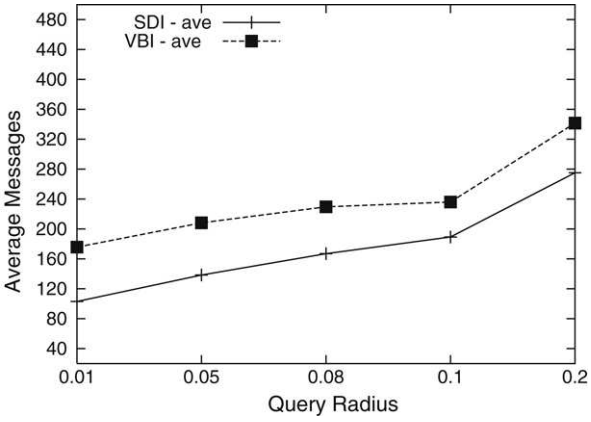


Fig. 12. Range query cost with different radius.

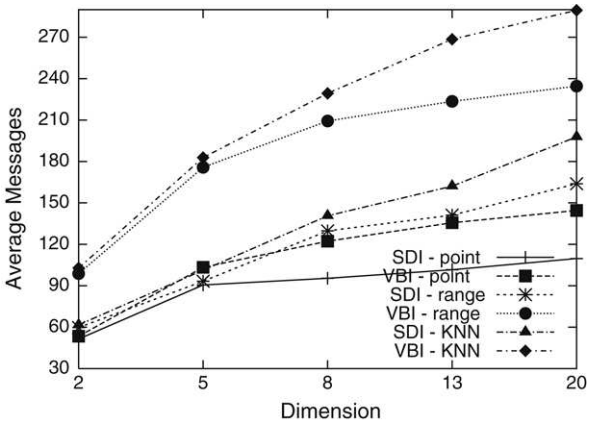


Fig. 13. Average query cost with different dimensionality.

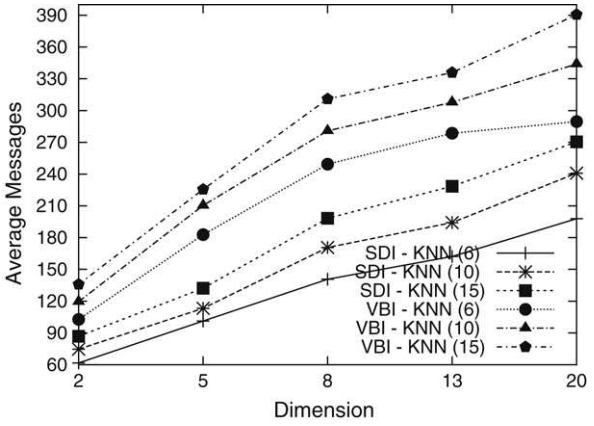


Fig. 14. KNN query cost.

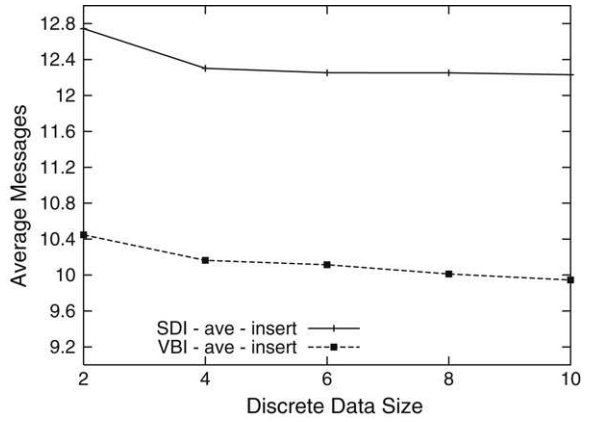


Fig. 15. Insertion cost with different discrete data size.

and KNN query, especially for a large network size, a large dimensionality data space, a very skewed data distribution or large discrete data size. The reason is that these elements affect the overlapping among nodes. The more the overlapping, the more advantage SDI gets, because we can use less delivering of messages to check the ancestor nodes.

7.2. Performance of insertion

Data insertion cost includes 3 parts: insert position locating, ancestor link updating and discrete data reinsertion. For SDI and VBI-Tree, the maximum number of messages to locate the insert position is $2\log N$ and $\log N$, respectively, but the insertion updating cost is bounded by $O(\log N)$ to ancestor link and $O(N)$ to upside path, respectively. Then, though VBI-tree can beat SDI in insertion their cost does not change by much, around 2 messages, as shown in Figs. 15 and 17, because VBI-tree will spend much more on updating to the upside path as shown in Figs. 16, 18 and 19. When increasing discrete data size, the updating cost will be lower and then the insertion cost will be reduced, as shown in Fig. 15. Fig. 17 indicates that the larger the network size, the more the cost of an average insertion. Though VBI-tree beats SDI in insertion, compared to the query cost reduction as shown in Fig. 9, it is acceptable. The larger the network size, the more the insertions or the more

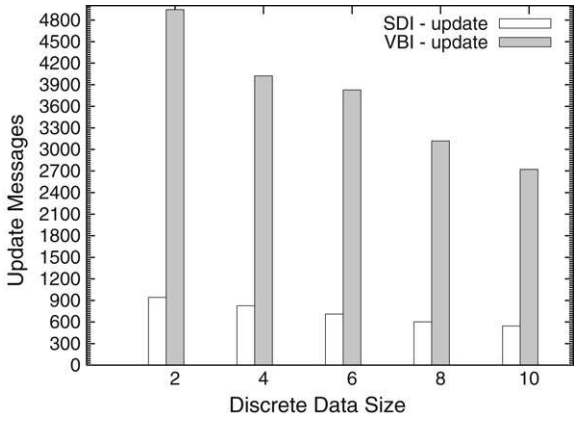


Fig. 16. Insertion update cost with different discrete data.

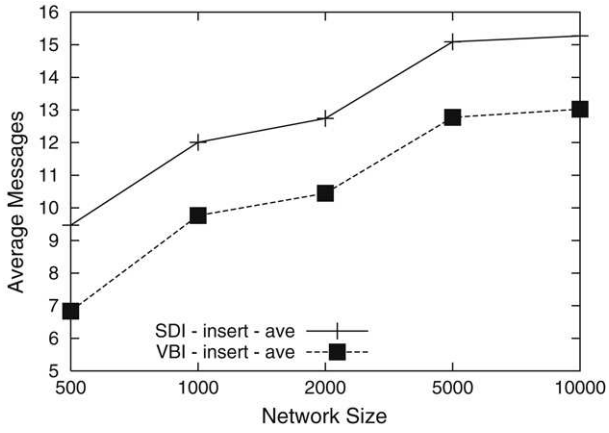


Fig. 17. Insertion cost with different network size.

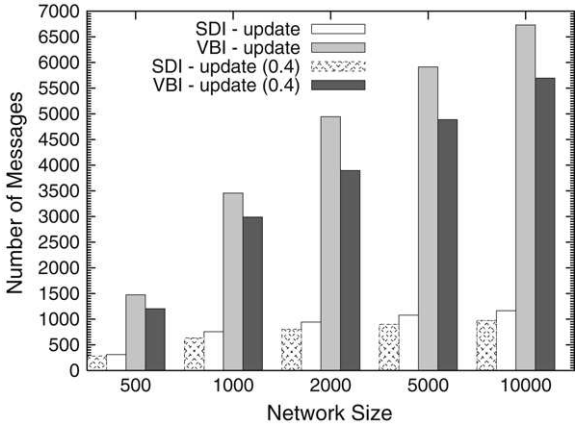


Fig. 18. Insertion update cost with different network size.

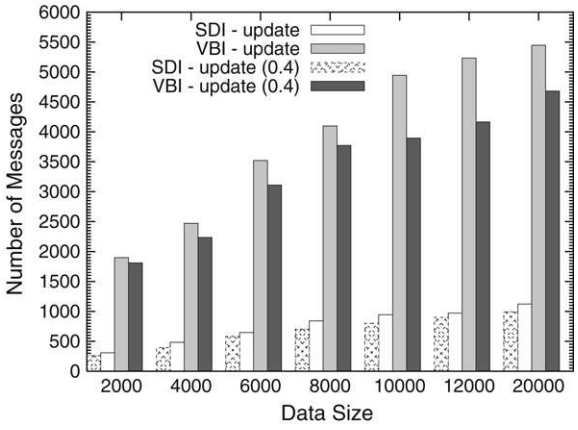


Fig. 19. Insertion update cost with different data size.

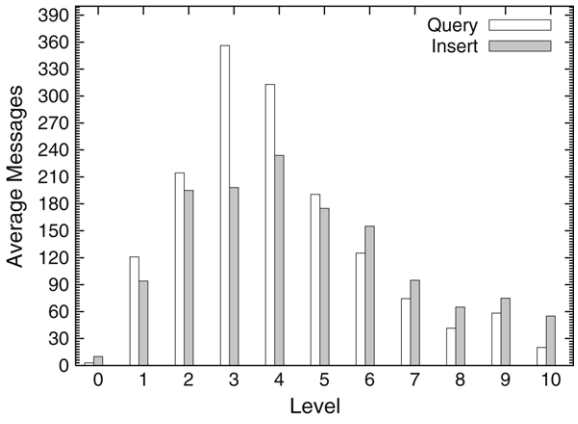


Fig. 20. Average query/insertion load at different levels.

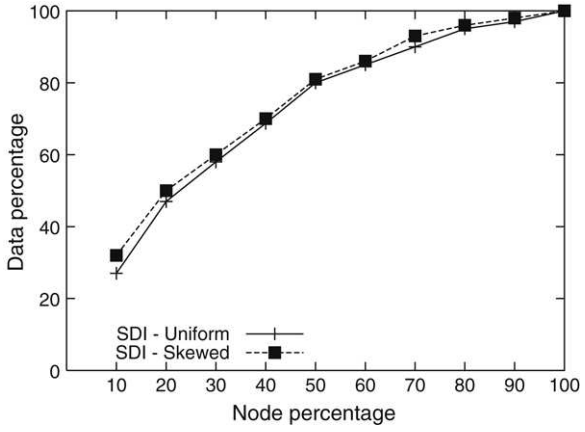


Fig. 21. Data distribution.

skewed the data distribution, the more the updating cost is reduced by SDI, as shown in Figs. 18 and 19. From these results, we find SDI will show great advantages if it mainly processes queries other than data operations (insertion or deletion).

7.3. Access load

Access load is measured by the average messages received by the nodes belonging to each level when doing data insertion and searching. Level 0 is the root and the highest numbered levels are LRNs. Fig. 20 shows that either when inserting or when query processing, the load does not concentrate on the root or nodes near the root. In our design, the request will always be processed among the same level neighbor nodes if the sideways routing tables are full. Then the middle levels will take more load because they have more nodes. Only when doing discrete data checking, do we send the query upwards to the lower level nodes. So in any case, there is no root-bottleneck problem in SDI. We compare the work load[7] distribution for uniform data and skewed data, as shown in Fig. 21. It indicates that SDI is scalable as it is not sensitive to the data distribution.

[7] It is common sense that a peer's query resolving load is proportionate to number of data it contains.

8. Conclusion

Indexing multidimensional data is an essential problem for bringing peer-to-peer technologies into mission critical data management applications. An enabling technique for this purpose should not only keep search efficiency in a static environment, but also provide availability and robustness without performance sacrifice in a large-scale distributed system where nodes may join or leave the system dynamically. We introduce SDI, a tree-based overlay network, in which each node only maintains carefully selected additional links to ancestor and descendant nodes. We show that even with less additional links, our search algorithms based on SDI still bounds the query efficiency by $O(\log N)$. The advantage achieved by the simple yet efficient index structure is huge. It eliminates the root bottleneck problem suffered by most other tree-based overlay networks. Furthermore, since less links are maintained, it reduces the cost for both network maintenance and query processing. Experimental results show that it scales well with the query range, network size, discrete data area size, data distribution and data dimensionality.

Acknowledgment

The work is partially supported by the National Natural Science Foundation of China under Grant No. 60496325.

References

[1] A. Andrzejak, Z. Xu, Scalable, efficient range queries for grid information services, in: P2P, 2002, pp. 33–40.
[2] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.
[3] A. Crainiceanu, P. Linga, J. Gehrke, J.S. Mugasundaram, Querying peer-to-peer networks using *p*-trees, in: WebDB, 2004, pp. 25–30.
[4] C. Schmidt, M. Parashar, Flexible information discovery in decentralized distributed systems, in: HPDC, 2003, pp. 226–235.
[5] C. Tang, Z. Xu, S. Dwarkadas, Peer-to-peer information retrieval using self-organizing semantic overlay networks, in: SIGCOMM, 2003, pp. 175–186.
[6] C. Zhang, A. Krishnamurthy, R. Wang, SkipIndex: Towards a scalable peer-to-peer index service for high-dimensional data, Technical Report Tr-703-04, Princeton University, 2004.
[7] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, B. Cantania, Indexing techniques for advanced database applications, Kluwer Academics, 1997.
[8] H.V. Jagadish, B.C. Ooi, Q.H. Vu, BATON: A balanced tree structure for peer-to-peer networks, in: VLDB, 2005, pp. 661–672.
[9] H.V. Jagadish, B.C. Ooi, Q.H. Vu, R. Zhang, A. Zhou, VBI-Tree: A Peer-to-Peer framework for supporting multi-dimensional indexing schemes, in: ICDE, 2006, pp. 34–43.
[10] J.K. Lawder, P.J.H. King, Using space-filling curves for multi-dimensional indexing, in: Advances in Databases, BNCOD 17, in: Lecture Notes in Computer Science, vol. 1832, 2000, pp. 20–35.
[11] J.L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (9) (1975) 509–517.
[12] D.A. Trana, T. Nguyenb, Hierarchical multidimensional search in peer-to-peer networks, FGCS 31 (2) (2008) 346–357.
[13] J. Lee, H. Lee, S. Kang, S.M. Kim, J. Song, CISS: An efficient object clustering framework for DHT-based peer-to-peer applications, Computer Networks 51 (4) (2007) 1072–1094.
[14] M. Batko, et al., Scalability comparison of Peer-to-Peer similarity search structures, FGCS (2007) doi:10.1016/j.future.2007.07.012.
[15] P. Ganesan, B. Yang, et al. One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems, in: WebDB, 2004.
[16] N. Koudas, C. Faloutsos, I. Kamel, Declustering spatial databases on a multicomputer architecture, EDBT (1996) 592–614.
[17] M. Li, W.-C. Lee, A. Sivasubramaniam, DPTree: A balanced tree based indexing framework for peer-to-peer systems, ICNP, 2006, pp. 515–532.
[18] A. Mondal, M. Kitsuregawa, B. Ooi, K. Tan, R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases, in: ACM GIS, 2001, pp. 28–33.
[19] A. Mondal, Y. Lifu, M. Kitsuregawa, P2PR-tree: An R-tree-based Spatial Index for Peer-to-Peer Environments, in: Current Trends in Database Technology - EDBT 2004 Workshops, 2004, pp. 516–525.
[20] O.D. Sahin, A. Gupta, D. Agrawal, A. Abbadi, A peer-to-peer framework for caching range queries, in: ICDE, 2004, pp. 165–176.
[21] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: VLDB, 1997, pp. 426–435.
[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: SIGCOMM, 2001, pp. 161–172.
[23] Y. Shu, K.-L. Tan, A. Zhou, Adapting the content native space for load balanced indexing, in: DBISP2P, 2004, pp. 122–135.
[24] D. Novak, P. Zezula, M-Chord: A scalable distributed similarity search structure, in: INFOSCALE, 2006, pp. 1–10.
[25] F. Falchi, C. Gennaro, P. Zezula, A content-addressable network for similarity search in metric spaces, in: DBISP2P, 2005, pp. 126–137.
[26] M. Batko, C. Gennaro, P. Zezula, Similarity grid for searching in metric spaces, in: DELOS Workshop, 2005, pp. 25–44.
[27] C. Doulkeridis, A. Vlachou, Y. Kotidis, M. Vazirgiannis, Peer-to-Peer similarity search in metric spaces, in: VLDB, 2007, pp. 986–997.
[28] M. Cai, M. Frank, J. Chen, P. Szekely, MAAN: A multiattribute addressable network for grid information services, in: GRID, 2003, pp. 184–191.
[29] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: Supporting scalable multi-attribute range queries, SIGCOMM Comput. Commun. Rev. 34 (4) (2004) 353–366.

Rong Zhang has just graduated from Fudan University. She received her M.S. degree in Computer Science and Technology from Northeastern University, Shen Yang, China in 2004. Now she is a researcher in National Institute of Information and Communication Technology, Japan. She has had papers published in SIGMOD, ICDE, INFOSCALE. Her research interests include peer-to-peer, data mining and graph data processing.

Weining Qian ia currently an associated professor of Institute of Massive Computing, Software Engineering Institute, at East China Normal University, Shanghai, China. Before this he was an Assistant Professor in Department of Computer Science and Engineering at Fudan University. He received his M.S. and Ph.D. degrees in computer science from Fudan University in 2001 and 2004. His research interests include data mining for very large databases, data stream query processing and mining and peer-to-peer computing.

Aoying Zhou is currently a professor and the director of Institute of Massive Computing at East China Normal University, Shanghai, China. He won his Ph.D. degree from Fudan University in 1993. He is now serving as the vice-director of ACM SIGMOD China and Database Technology committee of China Computer Federation. He was invited to join the editorial boards of some prestigious academic journals, such as VLDB Journal, Journal of Computer Science and Technology (JCST), etc. He served or in serving as PC members of ACM SIGMOD07/08, WWW07/08, SIGIR07, EDBT06, VLDB05, VLDB05, ICDCS05, and etc. He was the conference co-chair of ER04 and the PC con-chair of WAIM00. His research interests include massive data management, Web data management and Web mining, Web services, data streams, and P2P computing systems.

Minqi Zhou is a Ph.D. candidate in Fudan University, China. His research interestes include data management in P2P environment, such as preference search, publish/subscribe system.