SDI: A Swift Tree Structure for Multi-dimensional Data Indexing in Peer-to-Peer Networks

Rong Zhang
Computer Science and Engineering
Fudan University, China
rongzh@fudan.edu.cn

Minqi Zhou
Computer Science and Engineering
Fudan University, China
zhouminqi@fudan.edu.cn

Weining Qian
Software Engineering Institute
East China Normal University, China
wnqian@sei.ecnu.edu.cn

Aoying Zhou
Computer Science and Engineering
Fudan University, China
ayzhou@fudan.edu.cn

ABSTRACT

Efficient multi-dimensional data search has received much attention in centralized systems. However, its implementation in large-scale distributed systems is not a trivial job and remains to be a challenge. In this paper, SDI, a new succinct multi-dimensional balanced tree structure based on peer-to-peer technology, is presented. With SDI structure, the query efficiency can be bounded by $O(\log N)$. Compared with previous tree-based methods, SDI has extremely low maintenance cost. This is due to the carefully chosen finger links. Furthermore, new algorithms are designed for both point query and range query processing, which make SDI free from the root-bottleneck problem. Experimental results validate the efficiency and effectiveness of the proposed approach.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; H.2.4 [**Systems**]: Query processing; H.2.m [**Database Management**]: Miscellaneous

General Terms

Algorithms, Experimentation, Design

Keywords

multi-dimensional data, peer-to-peer, point query, range query

1. INTRODUCTION

Efficient multi-dimensional data search is a key challenge for data management in large-scale distributed systems. Existing solutions [18, 12, 5, 4, 3] are mostly based on dimensionality reduction or space partition. The performance of these methods falls drastically when dimensionality increases.

Tree-based multi-dimensional data indexing is widely used in centralized systems. However, it is non-trivial to apply those methods directly in distributed environments. First, tree-based method often suffers the root-bottleneck problem, since high-level[1] nodes tend to forward queries to low-level nodes. Second, as in centralized systems, space-division-based methods may result in space overlapping for non-uniform data which leads to intolerable huge number of messages for query processing in large-scale network environment.

As one state-of-the-art technology, Virtual Binary Index Tree (VBI-Tree) [9] was proposed, which is the first general framework to implement hierarchical space-division-based tree structures, such as R-tree [2] and M-tree [15], in large-scale distributed networks using Peer-to-Peer (P2P) technologies. It addressed all the problems of tree-based structure, including search efficiency, load balancing, fault recovery and root-bottleneck. However, in order to promise query efficiency, each node in VBI-Tree has to keep coverage information of all its ancestors, which results in additional overhead for network maintenance and data updating. Additionally, such information is not helpful dealing with "discrete data" checking, which affects the system scalability to large network size, highly skewed data distribution and so on.

In this paper, we present SDI, a Swift tree structure for multi-dimensional Data Indexing in P2P systems, inspired by VBI-Tree [9], but having less maintenance overhead and better query processing performance. Its novel features are listed as follows:

- SDI has a succinct tree structure with less additional network links. With the carefully designed routing links, they will not overload low-level nodes, such as the tree root, and the maintenance overhead is low.
- New point query and range 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.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

INFOSCALE 2007, June 6-8, Suzhou, China

Copyright © 2007 ICST 978-1-59593-757-5

DOI 10.4108/infoscale.2007.908

- SDI scales well in terms of query radius, network size and dimensionality.
- SDI still preserves the advantages of tree-based indexing techniques, including load balance, fault tolerant, and search efficiency.

Extensive experiments validate the efficiency and effectivenss 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 SDI structure. Section 5 explains the system building in detail. Section 6 describes query processing algorithms under the new architecture. In the end, Section 7 shows experimental study and Section 8 makes a conclusion.

2. RELATED WORK

Multi-dimensional indexing, including high-dimensional indexing, has received much effort in the centralized databases [6]. The representative ones are based on hierarchical space division, and then organized into trees, such as R-tree[2], R*-tree[13], M-tree[15], and so on. Dimensionality reduction is also used for multi-dimensional data indexing. Space Filling Curve[10] is a representative one, which in essence gives a linear ordering of all points in the data set. VA-file[16] and LPC-file[7] use the vector approximate method to overcome the curse of dimensionality brought in by dimensionality reduction. They try to filter the feature vectors to construct the approximate file for the original files, but have to read the whole approximation file for filtering.

In distributed networks, multi-dimensional indexing mainly bases on the methods referred above. CAN can be considered as the first system, which supports multi-dimensional data indexing. It is something like KD-tree [11], and can be used directly for indexing multi-dimensional data. [4, 1, 14] are all based on CAN. [1] uses the inverse Hilbert to map one dimensional data space to CAN. [14] is proposed for caching low-dimensional range queries. pSearch[4] is proposed for document retrieval in P2P networks. But it does not focus on range query or KNN query. [18, 12, 3] use Space Filling Curve for multi-dimensional data indexing. After that, a single dimension overlay will be used to index the transformed data. But they show poor performance when facing highly skewed data. SkipIndex [5] and DIM[17] are KD-tree based methods. SkipIndex [5] is the latest P2P system based on skip graph, which aims to support high-dimensional similarity query. But its scalability, failure recovery and query processing cost are problematic when facing high-dimensional data. DIM[17] supports multi-dimensional range queries in sensor networks by mapping multi-dimensional data space to a 2-dimensional space. Load balancing is not addressed in DIM. VBI-Tree[9] is the first general framework for multi-dimensional data indexing based on hierarchical tree structure with good performance. However, it has too many additional links which result in high cost in maintenance.

3. BACKGROUND KNOWLEDGE

In this section, we present a brief review of relevant features of VBI-Tree [9], which is a variation of BATON [8] and can be used to implement any kind of hierarchical tree structures that have been designed based on the space containment relationship. Each node, except the peer keeping the right most data node, in the network plays 2 roles: a data node and a corresponding routing node. We define **leaf routing node** (**LRN**) as the parent nodes of leaf(data) nodes. Data(leaf) node is used to store data and 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 child level is one level greater than parent level. Each level has at most 2^L nodes with L as the level number. We number 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). We put the neighbor routing nodes which have numbers less or greater than current node by a power of 2 in left or right routing tables. If there is no such neighbor node, we just put "null" in the corresponding position. A routing table is full if all valid neighbor positions are not null.

There are two important rules in VBI-Tree. The first one is a tree is balanced if every routing node which has a child has both it left routing table and right routing table full. The second is if a node x contains link to another node y in either its left or right routing table, the parent node of x must contain link to 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.

As we can see in VBI-Tree:

- Each node keeps an eye on all the ancestors which lie on the way to root, so each expansion or shrunk to space managed by any ancestor will cause all the descendants to update the corresponding uppath, reflecting the change to the ancestor. Updating is costly.
- There are only virtual links among descendants and ancestors, so discrete data checking message climbs up the tree one level one time, which costs many redundant messages before it reaches the target node.
- Routing nodes link data(leaf) nodes using adjacent links, which has mixed the structure and not restricted all the routing among routing nodes.

4. SDI ARCHITECTURE

In order to overcome the shortcomings of VBI-Tree, we put foward SDI, which is shown in Fig. 1. 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. **Adjacent link** is defined as the link which helps to connect all routing nodes, leaving alone

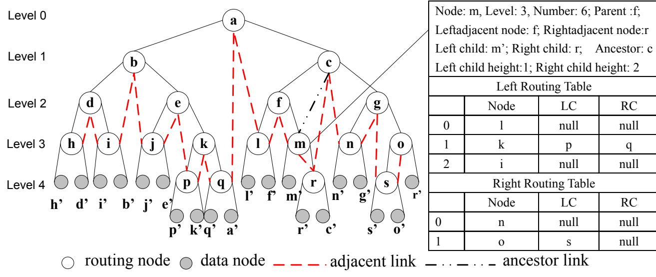

Figure 1: SDI Structure

the data nodes, by an in-order traversal as shown in Fig. 1. Given a node x, the node immediately prior or after to it, connected by the adjacent link, is the left or right adjacent node. Adjacent link always connects to one LRN at one end. **Ancestor link** is defined as the link distributed by the lower level routing node to its specific higher level routing node descendants, which are at least 2 levels higher[2] and lie on the left child branch but right most positions or the right child branch but the 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 size N. Child height is defined as the child subtree height, which is used to activate the balance algorithm [9]. To the data nodes, it has no sideways routing tables, ancestor or adjacent links, but only parent link to LRN.

In Fig. 1, 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 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 both LC and RC of l.

4.1 Ancestor Link Distribution

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

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

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 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 neither of them has ancestor link.

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

(2) Supposing (l_1, n_1) maintains the other ancestor link to (l_3, n_3). With Equation 1, we get

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

then we get:

$2 * n_2 - 1 = (2 * n_3 - 1) * 2^{l_2 - l_3}$ with $(l_2 >= 0, l_3 >= 0)$,

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) □

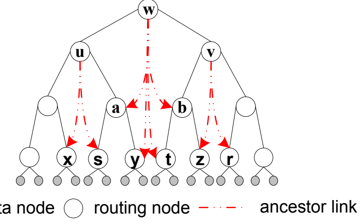

Figure 2: Ancestor Link Distribution

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 *2* nodes of the same level: the right(left) most descendant of the left(right) child branch. Let's take Fig.2 node w for example. At level 2 and level 3, it distributes the links to nodes a and y, which are the right most children of w's left branch, and nodes b and t, which are the left most children of w's right branch. Equation 1 and Theorem 1 help to decide whether one node has ancestor link and which level the ancestor locates, if any.

THEOREM 2. *Supposing there are nodes x, y, z at level l_1, evenly numbered (s, t, r oddly numbered) and sibling nodes u, v at level l_2, $(l_1 > l_2 + 1, l_2 > 0)$. Node w is the parent of node u, v. $x, y, z(s, t, r)$ are descendants of u, v, 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, z have links to u, w, 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, then $n_2 = 1/2 * n_1$ or else $n_2 = 1/2 * (n_1 + 1)$.
so $|N(y) - N(x)| = 2^{l_1 - l_2 - 1}$;
for x, y are of the same level, then x, y are neighbors.
So it does to node s, t, 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, z are of the same level, then node x, z are neighbors.
So it does to node s, t, r. □

Each node will keep the region information for the linked ancestor, if any. As shown in Fig. 2, 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. 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

Node join or departure process is similar with VBI-Tree [9] except the routing information updating because of the modification to routing tables. Currently one ancestor item is used to replace upside path and adjacent links do not relate to data(leaf) nodes any more. Only routing node is considered here.

After joining, if the new routing node is oddly numbered, but not the left most one, the ancestor link is 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 parent's original right adjacent link before updating. When joining, updating of the ancestor link can be piggyback over other updating messages, no additional cost is required. When node departure, if it is a node which can leave without causing the tree to become imbalanced[3], no ancestor link updating is needed, otherwise, at most additional $2\log N$ messages are used to do ancestor links updating for the descendants.

5.2 Index Building

Index building for SDI is the same as in VBI-Tree [9]. The ancestor node will cover the descendant node. Node joining or departure causes the data space divided or combined. Data adding or deleting causes the coverage space of node expanded or shrunk, which is the same process as in centralized index schemes. **Discrete data** is used to relax the updating cost, which is defined as the data belonged to no children temporarily.

5.3 Failure Recovery, Network Restructure and Load Balancing

Neighbor links and parent-child links are used to do failure recovery, the same as in VBI-Tree [9]. When dealing with load balancing, we can share load between parent and child for internal nodes, and between sibling nodes for data node. But for data nodes, we can also do lightly loaded node forceful joining to highly loaded node. When we meet with 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 changed much when doing subtree rotation, for the node position relations have not been changed. It means when do 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. Relying on the 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 by setting query radius to zero. For simplicity, we first consider the case where no sibling nodes overlap with each other. Later, we will show the general range query algorithm.

[3]It is the LRN with no neighbors having child routing node.

For region r_1 and region r_2, there are *3* different kinds of relationships between them: **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.

Algorithm 1: PointQuery(Node n, QueryPoint q, LowestLevel a, OldAncLevel oa,CheckedPath path)

```
1  begin
2    if path includes n then return ;
3    path.add(n);
4    if n covers q then
5      LocalSearch(n, q); SubtreeSearch(n, q);
6    else
7      if n.Level - a == 1 then
8        if n.Sibling!=null and n is not in path then
9          PointQuery(n.Sibling, q, n.Level, oa,
10           path);
11       else
12         LocalSearch(ancestors from
13         n.Parent.Level to oa, q);
14     else
15       if n is the left or right most node then
16         if n.Sibling !=null then
17           PointQuery(n.Sibling, q, a, oa, path);
18         else
19           PointQuery(n.Parent, q, a, oa, path);
20       else
21         if (anc = n.Ancestor) != null then
22           if anc covers q then
23             {leftChildM,rightChildM}=get
               maintainers for anc's children ;
24             if can not find 2 children
               maintainers of anc then
25               forward request to n.Parent ;
26             else
27               newOa=anc.Level;
28               if leftChildM isn't in path
                 then
29                 PointQuery(leftChildM,q,
30                 anc.Level+1,
                   newOa,path);
31                 newOa=anc.Level+1 ;
32               if rightChildM isn't in path
                 then
33                 PointQuery(rightChildM,
34                 q, anc.Level+1, newOa,
                   path);
35           else
36             if anc.Level == a and anc doesn't
               cover q and anc.Level > oa then
37               LocalSearch(ancestors from
                 n.Parent.Level to oa, q);
38             else
39               if anc.Level > a then
40                 y=anc.Parent.Maintainer ;
41                 if y! = null then
42                   PointQuery(y, q, a, oa,
43                   path);
44                 else forward the request to
45                   n.Parent;
46 end
```

6.1 Point Query Algorithm

We first present the point query algorithm with no sibling nodes overlapping with each other in Algorithm 1. The query forwarding stop conditions are :

- if one of its children can cover the query space completely, we can stop forwarding query upwards to lower level nodes.

- if sub-root of the checking branch can not intersect with the query space, we can stop forwarding 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 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.

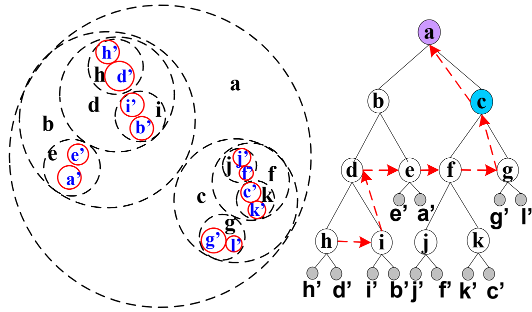

Figure 3: Point Query Processing

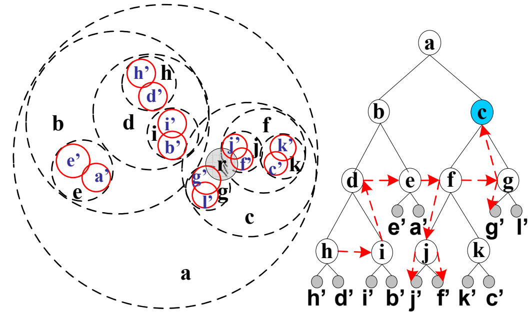

Figure 4: Range Query Processing

There are two cases when n has no ancestor reference. The first one happens to the tree or subtree less than 3 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.Level. Or else we do ancestor node discrete data search. The second one is that it is the left most or 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 next situation is that n can not cover q but it maintains an ancestor link[4]. If n's ancestor *anc* covers q and *anc*'s 2 children have not been checked, first we do the children checking before jumping directly to *anc* to do discrete data searching. By setting *oa* to *anc* position, it is easy

[4]Ancestor link contains ancestor's region information

Algorithm 2: RangeQuery(Node n, QueryRegion r, LowestLevel a, OldAncLevel oa,CheckedPath path)

```
1  begin
2      if path includes n then return;
3          path.add(n);
4      if n intersects r then
5          LocalSearch(n, r); SubtreeSearch(n, r);
6      if n.Level == a then return ;
7      if n.Level- a == 1 then
8          if n.Sibling! = null and n is not in path then
9              RangeQuery(n.Sibling, r, n.Level, oa, path);
10         if both n and n.Sibling don't cover r then
11             LocalSearch(ancestors from n.Parent.Level to
12             oa, r);
13         return;
14     if n is the left or right most node then
15         if n.Sibling! = null then
16             RangeQuery(n.Sibling, r, a, oa, path);
17         else
18             RangeQuery(n.Parent, r, a, oa, path);
19             return;
20     if (anc=n.Ancestor)!=null then
21         if anc intersects r then
22             {leftChildM, rightChildM}=get
               maintainers for anc's children ;
23             if can not find 2 children maintainers of anc
               then
24                 forward request to n.Parent;
25             else
26                 newOa=anc.Level ;
27                 if leftChildM isn't in path then
28                     RangeQuery(leftChildM, r,
                       anc.Level+1, newOa, path) ;
29                     newOa=anc.Level+1 ;
30                 if rightChildM isn't in path then
31                     RangeQuery(rightChildM, r,
32                     anc.Level+1, newOa, path);
33         if anc.Level ==a and anc doesn't cover r and
           anc > oa then
34             LocalSearch(ancestors from anc.Parent.Level
35             to oa, r) ;
36         if anc.Level > a then
37             if anc doesn't cover r then
38                 y=anc.Parent.Maintainer ;
39                 if y! = null then
40                     RangeQuery(y, r, a, oa, path);
41                 else forward the request to n.Parent;
42             else
43                 {newLevelList}=from anc.Parent.Level
                   to level a ;
44                 foreach newLevel in newLevelList
45                     z=(get a node on the other side of
46                     tree rooted at newLevel);
47                     RangeQuery(z, r, newLevel+1,
48                     newLevel+1, path);
49 end
```

to turn back to *anc* to do discrete data checking if no children cover q. Any missing of child maintainer will cause the parent node to forward the query. Later we restrict the legal checking level for the children by setting: *LowestLevel* to (*anc*.Level + 1) and *OldAncLevel* to *newOa*. The child with lower *newOa* value has the responsibility to activate the checking to the ancestor at *oa* level so as to avoid repeative discrete data search to ancestor at level *oa*. When none of *anc*'s children can cover q, we do discrete data checking at *anc*. Notice that all the discrete data checking can stop forwarding upwards once we meet with the node which can covers q.

The last situation is that 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 find the node covering the query.

In order to avoid receiving back the search request, we use parameters *CheckedPath* to log checked nodes, *LowestLevel* to mark the checking sub-root and *OldAncLevel* to mark lowest ancestor level which maybe need discrete data checking. Only if *LowestLevel* sub-tree covers r, nodes listed from *LowestLevel - 1* to *OldAncLevel* can be deleted from the checking list. The initial values for these *3* parameters are $null, 0, 0$ respectively.

Let's illustrate the point query algorithm in Fig. 3. This is a 2-dimensional M-Tree based space division. The query is issued at node h, and the target is 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 the 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 d and d will forward the request of finding a lower level ancestor to e. Here e finds that ancestor a covers q . According to the algorithm, before do a's discrete data checking, we shall send checking request to b's and c's maintainers, which are the children maintainers of a, at level 2. But b and c have no ancestor references maintained at this level. We choose the left and right direct neighbors, d and f, to forward the request. For d has been included in *path*, we forward the request to f. At f, it has *a=1* and *oa=0* which rooted at c. There is no ancestor reference distribution at this level. g is f's sibling and has not been visited before. Then it receives the search request. After doing local search, it begins to do ancestor nodes discrete data search to c and a. At last we get result at a.

In point query algorithm, query can be started at any position. 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 give out the point query processing algorithm without overlapping. We can not avoid overlapping, when using general hierarchical space division, especially for skewed data distribution. In such a case, query will be forwarded to multiple nodes instead of only one node. We will show the detail in range query processing algorithm.

6.2 Range Query Algorithm

Range query algorithm is shown in Algorithm 2 and it is similar with Algorithm 1. The main difference 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's say node n issues a range query r.

If n intersects with or covers r, we do local search (discrete data search) and also begin 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 query to all the other nodes which line on the other side of the subtree 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 covers or intersects with r, we have to rely on the ancestor reference to find the first node which intersects or covers r.

Let's illustrate the range query algorithm in Fig. 4. In this example, node h issues a range query r. At first, h checks itself locally. It neither intersects or covers r nor maintains any ancestor link. h forwards the request to its sibling i, which maintains 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 node d. It is the left most node and does not intersect with r. We forward the request to sibling e, which maintains the ancestor link to a. 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 direct neighbors in e's sideways routing tables. d has been included in *path.* We only forward request to f. Now the query are restricted to the branch rooted at node c, with *a=1* and *oa=0.* f begins the query processing in a shorter tree branch. It intersects with r. Its sibling g has not been checked before, so it receives the search request. But neither of them covers r, so we start checking on both g and f, and also forward discrete data checking request to ancestors. However a does no discrete data checking , because c covers r completely. At last, the query is resolved at peer *j, f, c* and *g.*

6.3 Analysis to Range Query Algorithm

Ancestor link plays the most important role in our algorithm.

1. As presented in Section 4.1, ancestor links are distributed evenly to the nodes of the same level and they contain ancestors' space coverage information. So checking to a lower level ancestor, whether it intersects or covers the query, can be resolved by the corresponding link maintainers (higher level descendants) and need not always be forwarded upwards. This solves the root-bottleneck problem.
2. Ancestor link helps to do discrete data search with less hops, which reduces the query processing cost drastically, compared with VBI-Tree. The advantage will be more obvious to larger network size. For example in Fig.1, if m needs do discrete data 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.
3. We promise all the nodes which intersect with the query can be visited once and only once.
4. The parallel query distribution will promise the query efficiency is $O(\log N)$.

7. EXPERIMENTAL STUDY

To evaluate the performance of our proposed system, 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 in the tree structure. Communication between nodes is via sockets. There is a fake sever which distributes events to peer nodes. We generate the inserted data by Zipfian method with parameter 1.0. Using SDI, we implemented M-tree [15]. VBI-M-Tree is used for comparison. Both systems deal with load balance and fault recovery similarly, for ancestor links in SDI and upside path in VBI-Tree have no effect on these aspects. Our comparisons focus on the cost for query processing, data operation, and also on load distribution. In this simulation environmental, the default values for network size, discrete data size, insert data size, dimen-

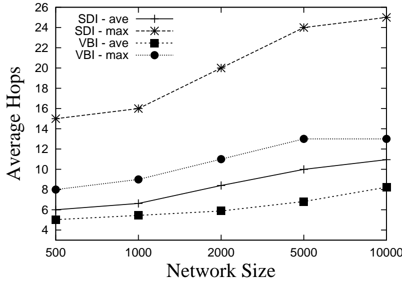

Figure 5: Average and Max. hops with different network size

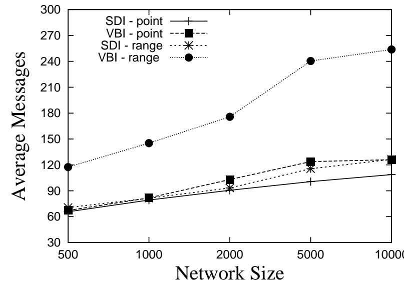

Figure 6: Average query cost with different network size

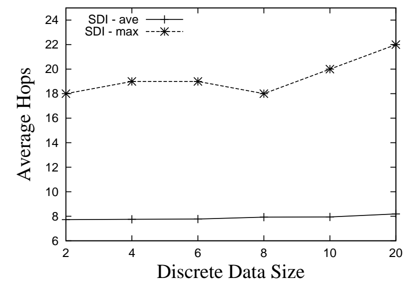

Figure 7: Average and Max. hops with different discrete data size

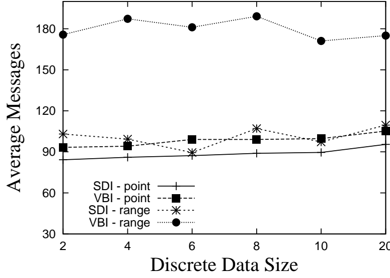

Figure 8: Average query cost with different discrete data size

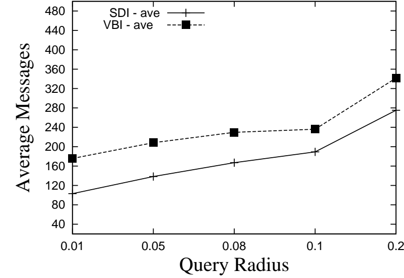

Figure 9: Range query cost with different radius

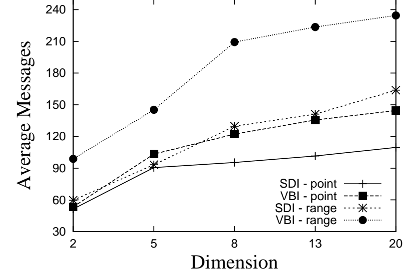

Figure 10: Average query cost with different dimensionality

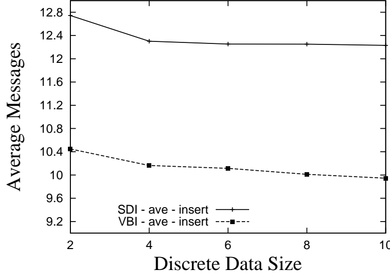

Figure 11: Insert cost with different discrete data size

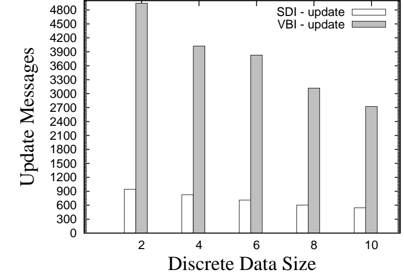

Figure 12: Insert update cost with different discrete data

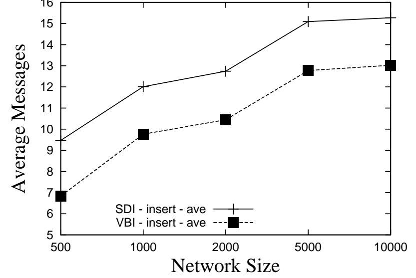

Figure 13: Insert cost with different network size

sionality, and query radius are $2,000$, 2, $10,000$, 5 and 0.01 respectively. The point or range query size is $1,000$.

7.1 Performance of Query Processing

Fig.5 shows that VBI-Tree beats SDI in both average and maximum hops to find results for the queries in different network size. VBI-Tree uses upside path to log the coverage information for all ancestors along the way to root and so each node has a wider view than SDI. Fig.6 shows the average messages to resolve a query in different network size. SDI beats VBI-Tree in both cases, especially for range query. As stated in Section 6.3, SDI resolves discrete data checking by using of ancestor links with less hops but VBI-Tree can only jump one level one time. When we increase of discrete data size, both 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 have not changed a lot, which are shown in Fig. 7 and Fig.8. Fig. 9 shows range query cost increases as we use bigger query radius. Fig.10 shows that query cost increases with increasing of dimensionality, for the higher the dimensionality, the more the space overlapping.

From the results, we can get VBI-Tree has better query processing efficiency than ours for its cost is bounded by $\log N$ and ours is bounded by $2logN$. However SDI beats VBI-Tree in average messages for resolving both point query and range query, especially for large network size, big dimensionality data space or big discrete data size. The reason is that these elements affect the overlapping among nodes. The more overlapping, the more advantage of our system, because we can use less delivering messages to check ancestor nodes.

7.2 Performance of Insertion

Data insertion cost includes 3 parts: data insertion, ancestor link updating and discrete data reinsertion. Fig.11 indicates that the larger the discrete data size, the less the average insertion cost. VBI-tree can beat SDI in insertion but their cost doesn't differentiate much, for VBI-tree will spend much more on updating for enlargement caused by discrete data as shown in Fig.12. Fig.13 indicates that the larger the network size, the more the cost on average insertion. But SDI will only cost about 2 more messages for insertion. Compared to the query cost reducing, it is acceptable. The bigger the network size or the more the insertion data size, the more the updating cost, as shown in Fig.14

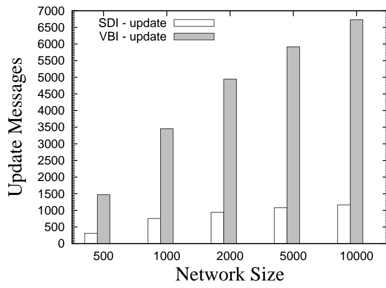

Figure 14: Insert Update Cost with different network size

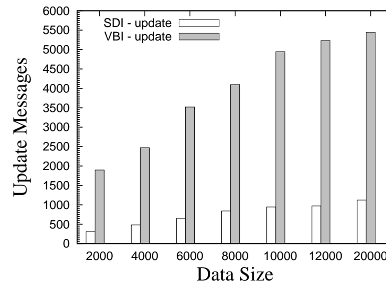

Figure 15: Insert update cost with different data size

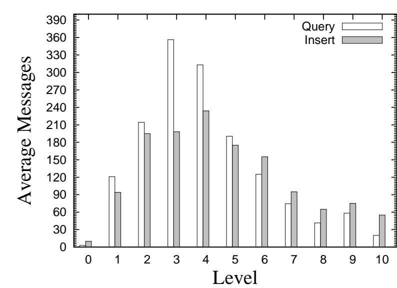

Figure 16: Average query/insert load at different levels

and Fig.15 respectively. SDI saves a large number of updating messages. For VBI-Tree, the insertion updating cost to upside path is $O(N)$, but SDI is $O(\log N)$ to ancestor link.

7.3 Access Load

It is measured by the average messages received by the nodes belonged to each level when doing data insertion and searching. Level 0 is the root and highest numbered level are LRNs. Fig.16 shows that either when insertion or query processing, the load does not concentrate on the root or nodes near root. In our design, the request will always be processed among the same level neighbor nodes if the sideways routing tables are full. Only when doing discrete data checking, we send the query upwards to the lower level nodes. So there is no root-bottleneck problem in SDI.

8. CONCLUSION

In this paper we have presented a swift general framework, SDI, for multi-dimensional query processing inspired by VBI-Tree. By defining specific link between an ancestor and the descendant, and query processing algorithms, we reduce cost for both network maintenance and query processing. The new query processing algorithms bound the query cost by $O(\log N)$ and avoid the root-bottleneck problem. SDI scales well to network size, discrete data cardinality and range query radius. In addition, SDI is much succinct since it does not maintain redundant adjacent links to data nodes, but keeps all the routing information among the routing nodes. The experiments indicate that SDI has achieved our desire successfully.

9. REFERENCES

[1] A.Andrzejak and Z.Xu. Scalable, efficient range queries for grid information services. *In P2P*, pages 33–40, 2002.

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

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

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

[5] C.Zhang, A.Krishnamurthy, and R.Wang. Skipindex: Towards a scalable peer-to-peer index service for high-dimensional data. *Technical Report Tr-703-04, Princeton University*, 2004.

[6] 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.

[7] G.-H.Cha, X.Zhu, D.Petkovic, and C.-W.Chung. An efficient indexing method for nearest neighbor searches in high-dimensional image databases. *IEEE Transactions on Multimedia*, 4(1):76–87, 2002.

[8] H.V.Jagadish, B.C.Ooi, and Q.H.Vu. Baton: A balanced tree structure for peer-to-peer networks. *In VLDB*, pages 661–672, 2005.

[9] H.V.Jagadish, B.C.Ooi, Q.H.Vu, R.Zhang, and A.Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. *In ICDE*, pages 34–43, 2006.

[10] J.K.Lawder and P.J.H.King. Using space-filling curves for multi-dimensional indexing. *In: Advances in Databases, BNCOD 17, Lecture Notes in Computer Science*, 1832:20–35, Jul. 2000.

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

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

[14] O.D.Sahin, A.Gupta, D.Agrawal, and A.Abbadi. A peer-to-peer framework for caching range queries. *In ICDE*, pages 165–176, 2004.

[15] P.Ciaccia, M.Patella, and P.Zezula. M-tree: An efficient access method for similarity search in metric spaces. *In VLDB*, pages 426–435, 1997.

[16] R.Weber, H.-J.Schek, and S.Blott. A quantitative analysis and performance study for similarity-search methods in high deimensional spaces. *In VLDB*, pages 194–205, 1998.

[17] X.Li, Y.-J.Kim, R.Govindan, and W.Hong. Multi-dimensional range queries in sensor networks. *In Sensys*, 2003.

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