Computer Communications xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries

Saeed Alaei [a], Mohammad Ghodsi [b,c,*], Mohammad Toossi [a]

[a] Computer Science Department, University of Maryland, College Park, USA
[b] Computer Engineering Department, Sharif University of Technology, Tehran, Iran
[c] School of Computer Science, Theoretical Physics and Mathematics, Tehran, Iran

A R T I C L E   I N F O

Article history:
Received 19 July 2008
Received in revised form 6 July 2009
Accepted 3 August 2009
Available online xxxx

Keywords:
Peer-to-peer networks
Distributed data structures
SkipNet
Range query

A B S T R A C T

This paper presents a new balanced, distributed data structure for storing data with multidimensional keys in a peer-to-peer network. It supports range queries as well as single point queries which are routed in $O(\log n)$ hops. Our structure, called SkipTree, is fully decentralized with each node being connected to $O(\log n)$ other nodes. We propose modifications to the structures, so that the memory usage for maintaining the link structure at each node is reduced from the worst case of $O(n)$ to $O(\log n \log\log n)$ on the average and $O(\log^2 n)$ in the worst case. It is also shown that the load balancing is guaranteed to be within a constant factor. Our experimental results verify our theoretical proofs.

1. Introduction and related work

Over the past few years, there has been a trend to move from centralized server based network architectures toward decentralized and distributed architectures and peer to peer networks. The term *Scalable Distributed Data Structure* (SDDS) first introduced by Litwin et al. in LH* [16] refers to this class of data structures which hold the following properties:

- There is no central directory.
- Client images (i.e. client information on where data is located) may be outdated, and is only adjusted in response to read queries.
- A client may send a request to an incorrect server, which will be forwarded to the correct one and the client image will be updated.

Litwin et al. modified the original hash-based LH* [16] structure to support range queries in RP* [15,16]. Based on the previous work of distributed data structures like LH* [16], RP* [15] and Distributed Random Tree (DRT) [12], new data structures based on either hashing or key comparison have been proposed like Chord [23], Viceroy [18], Koorde [10], Tapestry [25], Pastry [22], PeerDB [20], and P-Grid [1]. Most existing peer-to-peer (or P2P) overlays require $\Theta(\log n)$ links per node in order to achieve $O(\log n)$ hops for routing. Viceroy [18], Koorde [10], D2b [6], FissionE [17], and MOORE [7] which are based on DHTs, are the remarkable exceptions in that they achieve $O(\log n)$ hops with only $O(1)$ links per node at the cost of restricted or no load balancing. Family Tree [24] is the first overlay network which does not use hashing but supports routing in $O(\log n)$ hops with only $O(1)$ links per node.

Typically, the systems which are based on DHTs and hashing lack the range-query operation, locality properties and control over distribution of keys, due to hashing. In contrast, those based on key comparison, although requiring more complicated load balancing techniques, do better in these respects. P-Grid [1] by Aberer et al. is one of the systems based on key comparison which uses a distributed binary tree to partition a single dimensional space with network nodes representing the leaves of the tree and each node having a link to some node in every sibling subtree along the path from the root to that node. Gridella [2] a P2P system based on P-Grid working on GNutella has also been developed. Other systems like P-Tree [5] have been proposed that provide range queries in single dimensional space. Besides, some data structures like dB-Trees [9] based on B-Trees have been developed for distributed environments.

SkipNet [8] on which our new system relies heavily, is another system for single dimensional spaces based on an extension to skip lists. We basically extend SkipNet to handel multi-dimensional spaces.

G-Grid [21] is a solution proposed for the multidimensional case which is also based on partitioning the space into regions. However, regions in G-Grid are restricted in that they can only be split to two

Q1 * Corresponding author. Tel.: +98 21 6616 4625.
E-mail addresses: saeed.a@gmail.com (S. Alaei), ghodsi@sharif.edu (M. Ghodsi), mohammad@bamdad.org (M. Toossi).

0140-3664/$ - see front matter © 2009 Published by Elsevier B.V.
doi:10.1016/j.comcom.2009.08.001

regions of equal size. So, their boundaries cannot take arbitrary values and are restricted to multiples of their size. Their size are also restricted to negative powers of 2.

RAQ [19] is also another solution for the multidimensional case which incorporates a distributed partition tree structure to partition the space. Its network model is similar to that of the P-Grid [1]. Therefore, it requires $O(h)$ links at each node and routes in $O(h)$ hops where h is the height of the partition tree which can be of $O(n)$ for an unbalanced tree. Although it has been shown [3] that even for such unbalanced trees the number of messages required to resolve a query still remains of $O(\log n)$ on the average, if the links are chosen randomly, the number of links a node should maintain and the memory requirement at each node for storing information about the path from that node to the root still remain of $O(h)$ which is as bad as $O(n)$ for unbalanced trees.

In this paper, we propose a new efficient scalable distributed data structure called the *SkipTree* for storing the keys in multidimensional spaces. Our system uses a distributed partition tree to partition the space into smaller subregions with each network node being a leaf node of that tree and responsible for one of the subregions. In contrast to similar tree-based solutions, the partition tree here is used only to define an ordering between the regions. The routing mechanism and link maintenance is similar to that of SkipNet and is independent of the shape of the partition tree, so in general our system does not need to balance the partition tree (in fact, it has been shown [13] that such a tree cannot be efficiently balanced by means of rotation). Our system, maintains a SkipNet by the leaves of the tree in which the sequence of nodes in the SkipNet is the same sequence defined by the leaves of the partition tree from left to right. Handling a single key query is almost similar to that of an ordinary SkipNet while range queries are quite different due to the multidimensional nature of the SkipTree.

Briefly, our proposed structure supports point and range queries for n nodes holding k-dimensional data, in $O(\log n)$ hops, with high probability. For each node, we use $O(\log n)$ links to other nodes which may lead to $O(n)$ memory per node in the worst case. We propose modifications to the structures, so that the memory usage for maintaining the link structure at each node is reduced to $O(\log n \log\log n)$ on the average and $O(\log^2 n)$ in the worst case.

In Section 2 we explain the basic structure of the SkipTree including the structure of the partition tree, its associated SkipNet and the additional information to be stored in each node. In Section 3, the algorithms for single and range queries are explained. In Section 4, the procedure for joining and leaving the network is described. Some techniques for load balancing in SkipTree are discussed in Section 5. We modify the SkipTree structure in Section 6 to reduce the amount of information that needs to be stored in each node about the partition tree.

In Section 7, we present experimental results that verify our theoretical proofs. We also experimentally compare the performance of SkipTree and RAQ for point and range query operations. And finally, Section 8 concludes the paper.

A preliminary version of this paper appears in [4].

2. Basic skiptree structure

The distributed data structure used in the SkipTree consists of two parts. First, a *Partition Tree* is used to divide the search space among the nodes. This is described in Section 2.1. Then, as is shown in Section 2.2, nodes are linked together using a technique similar to SkipNet.

2.1. Space partitioning

We assume that each data element has a key which is a point in our k-dimensional search space. This space is split into n regions corresponding to the n network nodes. Let $S(v)$ denote the region assigned to node v, a node which is responsible for every data element whose key is in $S(v)$.

We use *Partition Tree*, a binary tree, to perform this assignment, and denote it by $\mathcal{T}$ throughout this paper. The tree consists of *internal nodes* and leaves. Only leaves in $\mathcal{T}$ represent the actual nodes in our overlay network. Each internal node of $\mathcal{T}$ has a corresponding section in the search space. Thus, we extend the definition of $S(v)$ to also denote the region assigned to an internal node v in this tree.

Assuming r is the root of our $\mathcal{T}$, $S(r)$ is always the whole search space. Each internal node then recursively splits its region into two smaller regions using a $(k-1)$-dimension hyperplane equation. That is, if an internal node v has two children, l and r, which are its left and right children respectively, $S(l)$ will be the portion of $S(v)$ located on one side of the hyperplane specified by v and $S(r$ will be the space to the other side. A sample partition tree and its corresponding space partitioning are depicted in Fig. 1.

For a network node u, which corresponds to a leaf in $\mathcal{T}$, we call the path connecting the root of the tree to u the *Principal Path* of u. We refer to the hyperplane equations assigned to the internal nodes of $\mathcal{T}$, that are on the principal path of a node u (including

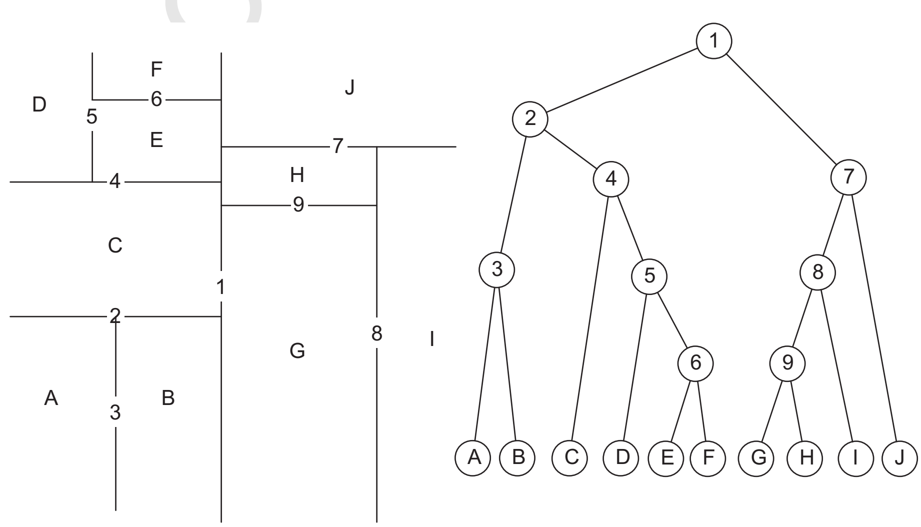

Fig. 1. A sample two dimensional partition tree, denoted by $\mathcal{T}$, and its corresponding space partitioning. Each internal node in $\mathcal{T}$, labeled with a number, divides a region using the line labeled with the same number. Each leaf of $\mathcal{T}$ is a network node responsible for the region labeled with the same letter.

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

information about on which side of those hyperplanes u resides) as the *Characteristic Plane Equations* of u or *CPE* of u for short. Every node in the SkipTree stores its own CPE as well as the CPE's of each of the $O(\log n)$ nodes (leaves) it has routing links to. Given a point p, and using the CPE information, every node u can locally identify if it holds p, or p belongs to a node to the left or the right of u or to the left or right of any other nodes it has links to (this becomes more clear when we explain the structure of link connections among the nodes.) The latter is useful in routing queries as explained in Section 3.

Storing the CPEs, however, requires $O(h \log n)$ memory at every leaf node, where h is the height of the tree. While this is of $O(\log^2 n)$ for a balanced tree, it may require as much as $O(n)$ memory in the worst case. We will provide a method for reducing the memory requirement in Section 6.

2.2. Network links

We link the network nodes in the SkipTree together by constructing a SkipNet structure among the leaves of $\mathscr{T}$ described in the previous subsection. However, using the SkipNet requires a total ordering to be defined on the nodes. We define this ordering to be the order in which the nodes appear as the leaves of $\mathscr{T}$ from left to right. We also make this sequence circular by considering the rightmost leaf to be on the left of the leftmost leaf and visa versa.

In an SkipTree in its ideal form, a node v keeps $2\log_2 n - 1$ links to other nodes. These are the 2^ith nodes to the left and right of v for every i from 0 to $\log_2 n$ as shown in Fig. 2. Unfortunately, maintaining this structure is very inefficient when handling node arrivals and departures. As a result, only an approximation to these ideal links is maintained in SkipNet.

For a given i, if we start from any node and follow the links that jump 2^i nodes in a specific direction in the ideal form, we will find a loop of length $n/2^i$. Let us call this loop an i-level ring. There are 2^i i-level rings. For example, there is only one 0-level ring, a circular doubly-linked list that connects every node in the aforementioned order. On the other hand, there are n last level rings consisting only of individual nodes. As illustrated in Fig. 3, the nodes in each i-level ring form exactly two disjoint $(i+1)$-level rings. This is the property that will be conserved when nodes are inserted into or deleted from the SkipTree in Section 4.

Finally, we note that a real number p_v is assigned to each node v. p_v is randomly generated when v joins the SkipTree so that $p_a < p_v < p_b$ where a and b are respectively v's predecessor and

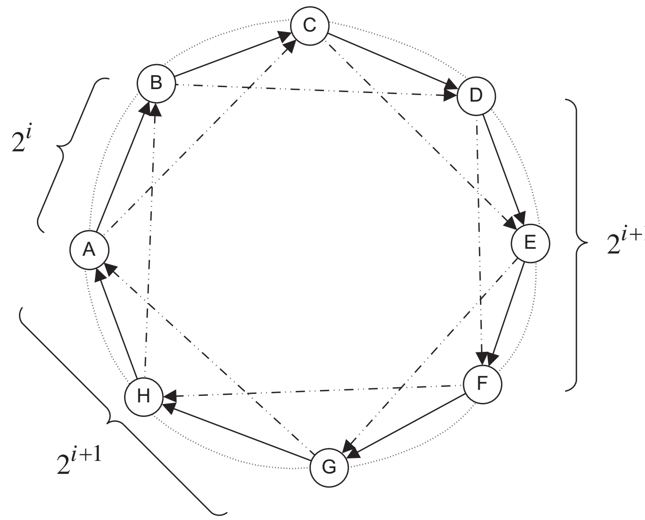

Fig. 3. The nodes in each i-level ring are split between two $(i+1)$-level rings. Solid arrows, representing i-level links, form the i-level ring. Dashed arrows are the next level links that form two disjoint $(i+1)$-level rings.

successor nodes in the total ordering. This number is used in Section 3.2 to handle range queries more efficiently.

3. Handling queries

Queries in a SkipTree can take two forms, either a single point query or a range query. We will discuss them separately on the following subsections.

3.1. Single point query

Whenever a node in the network receives a single point query, it must route it to the node which is responsible for the region containing that point. The routing algorithm is essentially the same as that used in the SkipNet. That is, every node receiving the query along the path, sends it through its farthest link that does not point past the destination node. This is shown in Fig. 4 where node S is about to route a single point query to some unknown node X which lies somewhere between node A and node B. Here, A and B are two consecutive nodes in the list of nodes to which A has direct links to

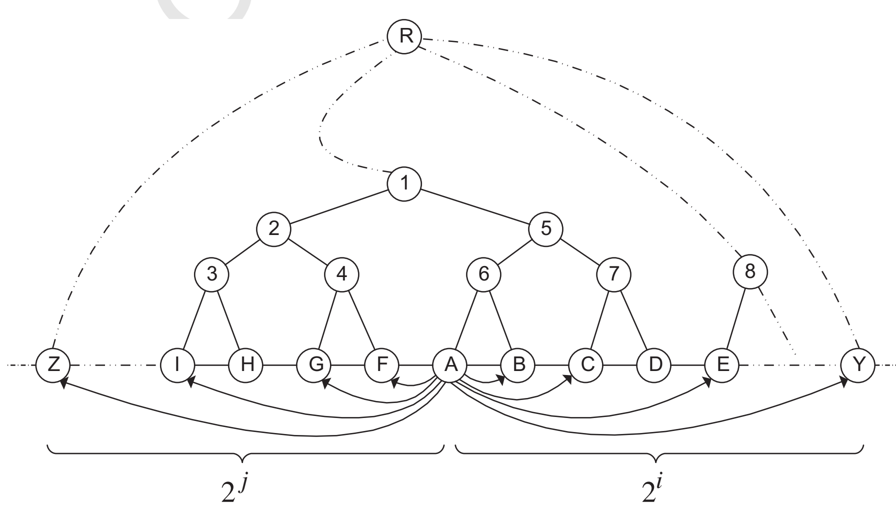

Fig. 2. The links maintained by node A in the ideal SkipTree. The target nodes are independent of the tree structure. The tree only helps us to put an ordering on the nodes. The *ith* link in each direction skips over $2^{i-1} - 1$ nodes in that direction.

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

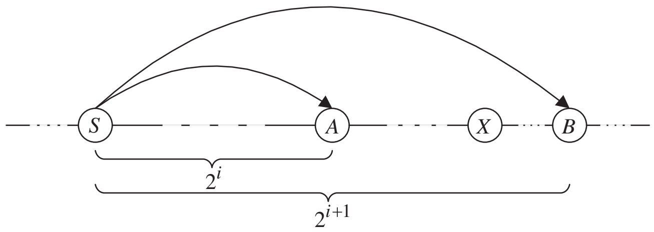

Fig. 4. A point query is routed through the farthest link which does not point past the destination node. Here, S receives a query targeting node X, so it routes the query to A. The distance to the destination node is at least halved at each hop.

and are respectively at distances 2^i and 2^{i+1} from A for some i. S routes the query to A and then A routes the query again in the same way until the it reaches its destination node. Note that the distance from A to the destination node is less than 2^i, so the next hop is at most 2^{i-1} nodes away from A. In fact, the distance of a query to the destination node is at least halved at each hop. This implies that the query reaches the destination after at most $\log_2 n$ hops. However, because SkipNet uses a probabilistic method for selecting and maintaining the links in the network, it guarantees routing in $O(\log n)$ hops w.h.p.[1] A formal proof of this can be found in [8].

In order that above procedure is effective, node S receiving a query point q must be able to identify whether the unknown node responsible for q lies to its left or to its right side. This is where $\mathcal{T}$ helps: S compares q against the planes in its CPE in the order they appear, starting from the root until it finds the first plane where the current node (u) and q lie on different sides of the plane. This is where we know that q is contained in a region belonging to the sibling subtree of u. If that subtree is a left (right) subtree, all of its nodes as well as the node containing q must also be to the left (right) of u. That is why every node in the network must also store the CPE of its link nodes in addition to its own CPE to be able to compare queries against its links too.

This procedure leads to $O(min(h \log n, n))$ memory usage at each node for storing the CPE, where h is the height of the tree. This may be as bad as $O(n)$ memory for an unbalanced tree. We will modify the tree structure in Section 6 to overcome this problem and guarantee $O(\log h \log n)$ memory usage at each node for the storage of CPE, which means $O(\log n \log \log n)$ on the average and $O(\log^2 n)$ memory usage in the worst case for an unbalanced tree.

3.2. Range query

A range query in the SkipTree is a 3-tuple of the form (R, f_s, l_s) where R is the query range and f_s and l_s are two real numbers which define the range of nodes in the sequence of nodes to be searched. That is, only the network nodes whose sequence numbers reside in the interval $[f_s, l_s]$ are searched. Using this form of queries, one can perform a complete range query for a region R using the 3-tuple $(R, -\infty, +\infty)$, so that all of the nodes are included in the search regardless of their sequence number. Note that the region defined by R can be of any shape as long as every node can locally identify whether R intersects with its given assigned region.

Handling a range query is very similar to that of a single point query with some minor differences. Suppose that a node S receives a range query (R, f_s, l_s). To handle this query, S breaks the range query to several (at most $O(\log n)$) new queries each targeting fewer number of nodes. A range query is propagated to each of the links maintained by S if there is node between that link and the next link that intersects with R. In other words, assume that A and B shown in Fig. 5 are two nodes corresponding to two consecutive links maintained by S. S sends a copy of the query to A if there is a node between A and B that intersects R. Every such node, if any, must reside in one of the crosshatched subtrees illustrated in the figure. In fact, such a node must be to the right of the nodes marked with + in $\mathcal{T}$ and to the left of the node marked with $-$. Because S has all of CPEs corresponding to its links, it also has the plane equations corresponding to the internal nodes marked with a + or $-$ sign. So, it can easily identify from those equations the regions in the multidimensional space associated with each of the subtrees between A and B. From this information, it can determine whether there is any subtree between A and B whose region intersects with R. If there exits such a subtree, it must also contain a node whose region intersects with R. In this way, a query is broken up by S to several queries and is propagated until it reaches its targets.

Note that the f_s and l_s fields of the query can be modified appropriately before a copy of the query is sent through a link. The reason is to restrict the sequence of nodes to be searched to prevent duplicate queries. For example in Fig. 5, suppose that a copy of the form (R, f_s, l_s) is to be sent from S to A. Also assume that $A.seq$ and $B.seq$ are the sequence numbers of A and B respectively. Then, S computes the interval $[f'_s, l'_s]$ as the intersection of $[f_s, l_s]$ and $[A.seq, B.seq]$ and it sends the query (R, f'_s, l'_s) to A. This will ensure that no nodes in the network receives the query more than once.

For the above procedure, note that the length of the path that a query travels through is of $O(\log n)$ regardless of the width of propagation at each hop. The proof is basically the same as in the single point query.

It is worth mentioning that this is the only place where sequence numbers are actually used. Sequence numbers make it possible to determine the relative ordering of two nodes of the network without knowing their corresponding plane equations or the regions they represent.

4. Node join and departure

Join and Departure operations are described in the following subsections. For each operation the node has to perform two relatively independent actions. Update $\mathcal{T}$ and update the network connections.

4.1. Joins

To join the SkipTree, a new node v has to be able to contact an existing node u in the SkipTree. v then splits the space assigned to u using a new plane. This allows u to transfer the control of one of the new regions along with its stored data items to v.

The algorithm is shown in function `v.join(u)` of Fig. 6. In lines 1 and 2, the new node copies the CPE of the existing node. Then, a new plane equation is generated to split the region formerly assigned to u. This plane can be arbitrarily chosen as our load balancing protocol will gradually change the partitioning to a more balanced configuration. Each of the two nodes then selects one of the two newly created regions. This is done in lines 3–5 by extending the principal paths using the `add_to_cpe ()` function.

The provided algorithm inserts v immediately after u and chooses the side of the plane containing the origin (ORIGIN_SIDE) for u and the other side (OTHER_SIDE) for v. This will help us in performing memory optimization in Section 6.

After updating $\mathcal{T}$, v has to establish its network connections. This is done by inserting v into approximately $\log n$ rings mentioned in Section 2.2. Starting with the level 0 ring, v randomly selects one of the two level $i+1$ rings derived from the selected level i ring until it reaches a ring in which there is no node except the

[1] An event is said to be occurring with high probability (w.h.p) if for any constant value of α the event occurs with a probability of at least $1 - O(\frac{1}{n^\alpha})$.

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

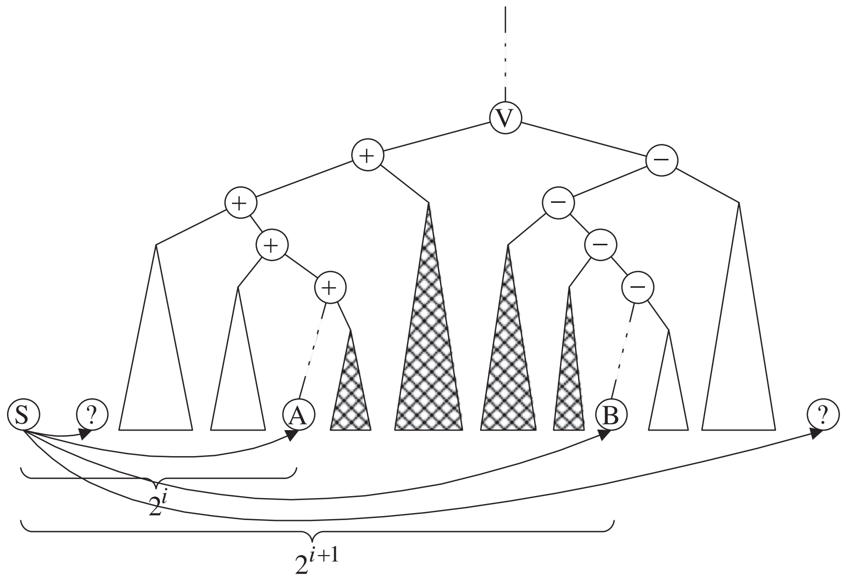

Fig. 5. A range query is propagated through each of the links maintained by *S* whenever there is a node which intersects *R* between that link and the next link. Here, *S* has consecutive links to *A* and *B*. A copy of the query is propagated to *A* if the subregion of any of the nodes between *A* and *B* intersects with *R*.

```
    //Join new node v to the SkipTree.
    //u is an arbitrary node in SkipTree.
    v.join(u){
1       v.cpe ← u.cpe;
2       v.height ← u.height;
3       new_plane ← choose_plane(v.cpe, v.height);
4       u.add_to_cpe(new_plane, ORIGIN_SIDE);
5       v.add_to_cpe(new_plane, OTHER_SIDE);
6       v.seq ← random number in (u.sequence, u.successor.sequence);
7       v.join_SkipNet(u);
8       u.transfer_data(v);
9       u.update_cpe(u.links);
    }
    //Append a plane to the CPE.
    v.add_to_cpe(plane, side){
        v.cpe[v.height].plane ← plane;
        v.cpe[v.height].side ← side;
        v.height ← v.height + 1;
    }
```

Fig. 6. Joining the SkipTree.

new node itself. v then moves backward, inserting itself in each of these rings with regard to the total ordering defined in Section 2.2 which is the same as using p_v. The exact algorithm is described in [8] and involves only $O(\log n)$ steps w.h.p.

Finally, u transfers the data items which are no longer in its assigned region to v in line 8. It also needs to send its new CPE to the nodes that have links to u. This is done in line 9 using the doubly connected links from u.

4.2. Departures

When node v is leaving the SkipTree, it has to follow three steps. First, update the poartition tree; second, transfer its data items to the appropriate nodes; and third, leave the SkipNet.

Suppose that v is responsible for region R and that the nodes in its sibling subtree are collectively responsible for the region S. In other words, the last plane in the node v's CPE, called P, splits its parent's region into regions R and S. To update $\mathscr{T}$, node v sends a special range query to the nodes in region S and instructs them to remove the plane P from their CPE. This will effectively remove v from $\mathscr{T}$ and shift every node in S one level closer to the root by removing their common parent.

To transfer the data items, v can simply find the node responsible for each item using a single point query and transfer the item accordingly. However, a more efficient method is to create a collection of the regions associated with every possible target node for v's data items and perform the single point queries for these items locally. This collection can be created by asking every node (as part of the previous special range query) in S to send its newly associated region to v if this region intersects with R.

In the last step, v has to close its $O(\log n)$ connections. As [8] points out, all pointers except the ones forming the level 0 ring can be re-

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

garded as redundant routing optimization hints and can be updated using a background repair process similar to Chord and Pastry. Therefore, v only needs to cleanly remove itself from the level 0 ring before leaving the SkipTree.

5. Load balancing

Many distributed lookup protocols use hashing to distribute keys uniformly in the search space and achieve some degree of load balance. Hashing cannot be used in the SkipTree as it makes range queries impossible. As a result, a load balancing mechanism is necessary to deal with the non-uniform key distribution.

Our load balancing protocol is derived from the *Item Balancing* technique in [11]. Load balancing is achieved using a randomized algorithm that requires a node to be able to contact random nodes in the network. This can implemented either using the existing network connections in SkipNet or using the underlying P2P routing framework. The second approach is preferred because of its higher speed and lower network traffic.

Let l_i, the load on node i, be the number of data items stored on i and α be a constant number so that $\alpha > 1$. We will prove that the SkipTree's load will be balanced w.h.p. if each node performs a minimum number of *load balancing tests* as per system *half-life*.[2]

Load Balancing Test: In a load balancing test, node i asks a randomly chosen node j for l_j. If $l_j \geqslant \alpha l_i$ or $l_i \geqslant \alpha l_j$, i performs a *load balancing operation*.

Load Balancing Operation: Assume w.l.o.g that $l_i < l_j$. First, node i normally leaves the SkipTree using the algorithm given in Section 4.2. Then, i joins the network once again at node j and selects a hyperplane for the newly created internal node in $\mathcal{T}$ in a way that the number of data elements is halved at both sides of the hyperplane. This makes both l_i and l_j to become equal to half the old value of l_j.

Theorem 1. [11] *If each node performs $\Omega(\log n)$ load balancing operations per half-life as well as whenever its own load doubles, then the above protocol has the following properties where N is the total number of stored data items.*

- With high probability, the load of all nodes is between $\frac{N}{8\alpha n}$ and $\frac{16\alpha N}{n}$.
- The amortized number of items moved due to load balancing is $O(1)$ per insertion or deletion, and $O(N/n)$ per node insertion or deletion.

The proof of this theorem using potential functions can be found in [11].

6. Memory optimization

Throughout the previous sections we assumed that every node in the network must store the CPE of each of $O(\log n)$ node to which it maintains a link to, as well as its own CPE. As we mentioned earlier, in a SkipTree of height h, this requires $O(h \log n)$ memory for each node. So, in an unbalanced SkipTree, a node may require $O(n)$ memory in the worst case. In this section, we enforce some constraints on the plane equations that a node may choose when joining the network and splitting another node, so that for a SkipTree of height h only $O(\log h)$ of the plane equations of any CPE will be needed.

The constraints that we enforce are the following, We assume that our search space is k-dimensional represented by $(x_1, x_2, \ldots, x_k)$.

1. Each plane must be perpendicular to a principal axis. That is, it must take the form of $x_i = c$ for some $1 \leqslant i \leqslant k$ and some value of c. This effectively means that every such plane partitions the keys in the a subspace based on the value of x_i for some i. We put further constraints that a plane $x_i = c$ associated with an internal node of u partitions a $S(u)$ into two smaller region such that the region containing the points with $x_i \leqslant c$ is assigned to the left subtree of u and the region containing the points with $c < x_i$ is assigned to the right subtree of u.
2. In this constraint, we precisely define the plane equation that is assigned to an internal node depending on the depth of that node. To do this, we first introduce the following notations for a node A in the SkipTree. Fig. 7 depicts some examples of these notations for $k = 2$.

   d_A: *depth of A:* the length of the principal path corresponding to A plus one.

   l_A: *level of A:* $l_A = \left\lceil \log_2\left(\frac{d_A}{k} + 1\right)\right\rceil$. This means that all nodes with depths in the interval $[k(2^i - 1) + 1, k(2^{i+1} - 1)]$ belong to level $i + 1$. This implies that on any principal path, the first k nodes are in level 1, the next $2k$ nodes are in level 2, the next $4k$ are on the next level and so on.

   d'_A: *relative depth of A:* is defined so that $d'_A = d_A - d_B + 1$ where B is the highest node which has the same level as A, or alternatively we can define $d'_A = d_A - k(2^{l_A - 1} - 1)$.

   s_A: *section number of A:* $s_A = \left\lceil \frac{d'_A}{2^{l_A - 1}} \right\rceil$. This imply that nodes at each level are partitioned to k sections: on the ith level of any principal path, the first 2^{i-1} nodes have section number 1, the next 2^{i-1} nodes are in Section 2 and so on.

   We are now ready to state the second constraint:If A is an internal node, the plane equation assigned to A must be of form $x_{s_A} = c$ for an arbitrary value of c. That is, for any given i, all of the nodes with section numbers of i are assigned plane equations of the form $x_i = c$. This implies that whenever a new node joins the SkipTree and splits the region of another node, which leads to a new internal node of say u, the plane equation u must obey the above schema. The only parameter that the new node can define to balance the load, when it splits a region, is the value of the constant c which should be enough for that purpose.A typical 2-dimensional space partitioned under the above constraints and its associated tree are shown in Fig. 8.

Lemma 1. *In any principal path of length h, nodes are partitioned to at most $k\lceil \log_2(\frac{h}{k} + 1)\rceil$ different sections.*

Proof. Since we defined the level of a node at depth d to be $\lceil log_2(\frac{d}{k} + 1)\rceil$, nodes in any principal cannot be partitioned to more than $\lceil log_2(\frac{h}{k} + 1)\rceil$ levels. Nodes at each level are further partitioned to k sections so there can be at most $k\lceil log_2(\frac{h}{k} + 1)\rceil$ sections in any principal path. □

Lemma 2. *For any leaf node A in a SkipTree, A needs to store only two plane equations for each section of its principal path.*

We call the sequence of the pairs of plane equations that node A stores, the *Reduced-Characteristic Plane Equations* of node A or for short the *RCPE* of node A.

Proof. All of the planes on the same section partition the space based on the value of the same field x_i. For example in Fig. 8, in Section 1 in the 3rd level of principal path of A, all of the internal

[2] A half-life is the time it takes for half the nodes or half the items in the system to arrive or depart [14].

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

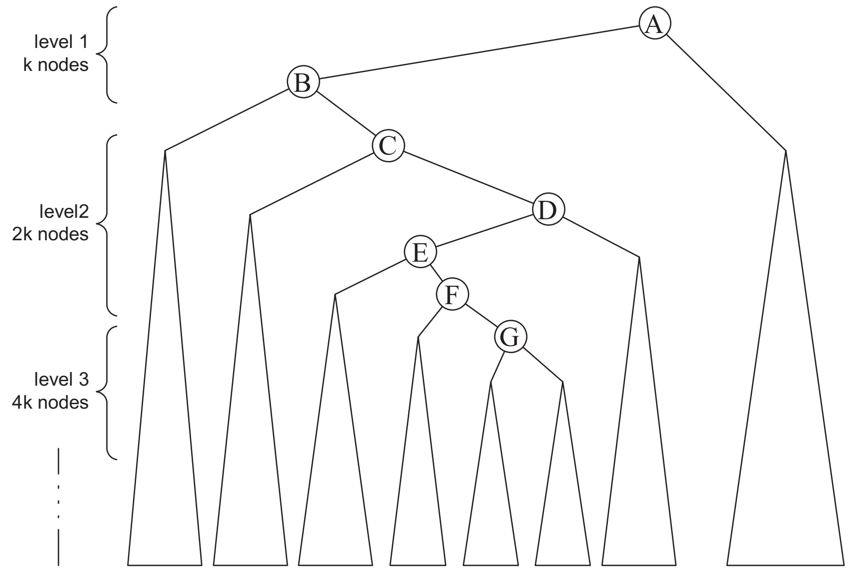

Fig. 7. A sample partition tree for a two dimensional space ($k = 2$). Nodes A to G have depths 1 to 7 respectively. A and B are on level 1; C, D, E and F are on level 2 and G is on level 3. The relative depth are: $d'_A = 1$, $d'_B = 2$, $d'_C = 1$, $d'_D = 2$, $d'_E = 3$, $d'_F = 4$, $d'_G = 1$.

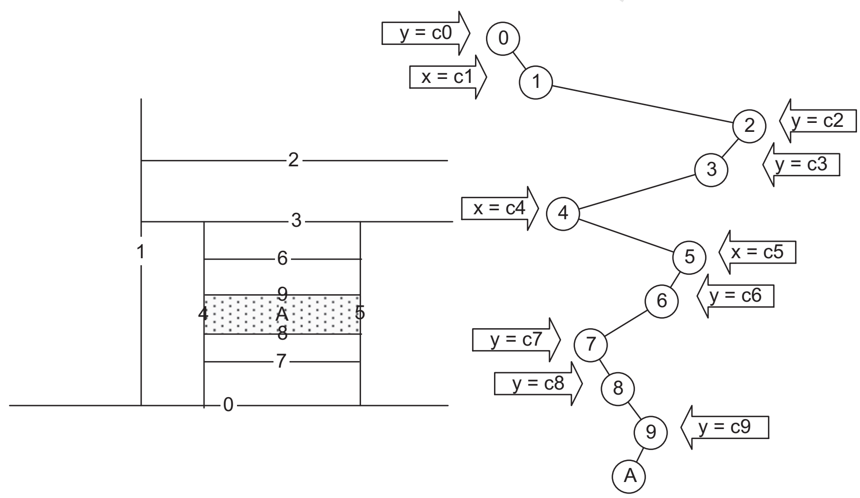

Fig. 8. The left is a sample partitioning of a 2-dimensional space under the memory optimization constraints, from the view point of node A and the right is the principal path of node A. The plane equations assigned to the internal nodes are shown in the arrows. Here, plane associated to x_1 is assumed to be of form $y = c$, and that for x_2 is of form $x = c$. The nodes 0 and 1 are in level 1. Therefore, their planes are $y = c_0$ and $x = c_1$ respectively. Nodes 2, 3, 4, and 5 are in level 2; so the planes of the first 2 are in $y = c$ form and those for the last 2 are of form $x = c$. Nodes 6–9 are all in level 3 (from the total of 8), so their planes are all of form $y = c$

nodes are assigned a plane equation of the form $y = c$ for different values of c and the region associated with node A or any other leaf for that matter is between at most two of the planes of that section. That is, for every section in the principal path for any node A, there are at most two planes which best represent the region in which $S(A)$ lies. So for each of the sections, A needs to store an inequality of the form $a \leqslant x_i < b$. Therefore, an RCPE can be stored as an ordered sequence of inequalities of the form $a \leqslant x_i < b$, one for each section in the principal path. □

When a node like A receives a point query, it finds the first inequality in the RCPE sequence that does not hold for the queried point. Then, the first constraint introduced above, ensures that the destination node which is responsible for the queried point will be to the left of the current node, if the point is to the left of the interval represented by the first unsatisfied inequality, or to the right of the current node otherwise. The situation with range queries is quite similar.

The sequence of inequalities in the RCPE for the node A in Fig. 8 is shown bellow:

- level = 1, section = 1: $c_0 \leqslant y < +\infty$
- level = 1, section = 2: $c_1 \leqslant x < +\infty$
- level = 2, section = 1: $c_0 \leqslant y < c_3$
- level = 2, section = 2: $c_4 \leqslant x < c_5$
- level = 3, section = 1: $c_8 \leqslant y < c_9$

6.1. Node join and departure

Joining mechanism is the same as what we described before, except that a new node must obey the above mentioned constraints. However leaving is a bit tricky since when some node A is about to leave, it must remove the internal node v which is its direct parent. This causes the plane associated with v to be removed from the RCPE of all nodes in its sibling subtree as well. This causes a problem only when v does not belong to the last level in the principal path of some node B in its subtree. In such case, removing v will contradict the second constraint of memory optimization we mentioned above.

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

To overcome this problem, A sends a special form of range query containing the region associated with its sibling (u) subtree (refer to Fig. 9.) By this special query it tries to find some node X in u's subtree such that the sibling subtree of X does not contain a node whose level is greater than that of X. Such a node X must always exist. For example the lowest node in the u's subtree of A always has the desired property. A will choose the left-most node with such property. Now, A and X swap their roles. That is, they swap their associated regions as well as the keys that they store and their position in the SkipTree. After this swapping, A will be in place of the X in the SkipTree so it can then leave the network with no problem, using the procedure described in the previous sections.

6.2. Complexity

The memory requirement of any node A for storing its RCPE as well as the RCPE of its links as described earlier is of $O(\log h \log n)$ where h is the height of the tree which is a major improvement over the $O(\min(h \log n, n))$ memory requirement in the default case.

In addition to the memory requirement guarantee, the constraints that we enforced in Section 6, guarantee the following strong invariant on the distribution of planes in each direction for every principal path:

Theorem 2. *For every principal path in a SkipTree if m_i is the number of plane equations of the form $x_i = c$ and m_j is the number of plane equations of the form $x_j = c$ for possibly different values of c, then the inequality $m_i \leqslant 2m_j + 1$ must always hold.*

Proof. For every principal path in a SkipTree, there are equal number of plane in each direction at each level except possibly for the last level (the level with highest number, that is the level of the lowest part of the path), because every level except the last level consists of exactly k different sections of equal size with all of the plane of each section being in the same direction. Besides, the number of planes in a single section at the *ith* level is 2^{i-1}, so if a principal path consists of r levels, for each direction, the total number of planes in that direction at all levels except the last level is $2^0 + 2^1 + 2^2 + \cdots + 2^{l-2}$ which is $2^{l-1} - 1$. Also, for each direction, the number of planes in that direction at the last level is between 0 to 2^{l-1}. So, in any principal path, for each direction, the total number of planes in that direction at all levels is between $2^{l-1} - 1$ to $2^l - 1$ and the above inequality obviously results. $\square$

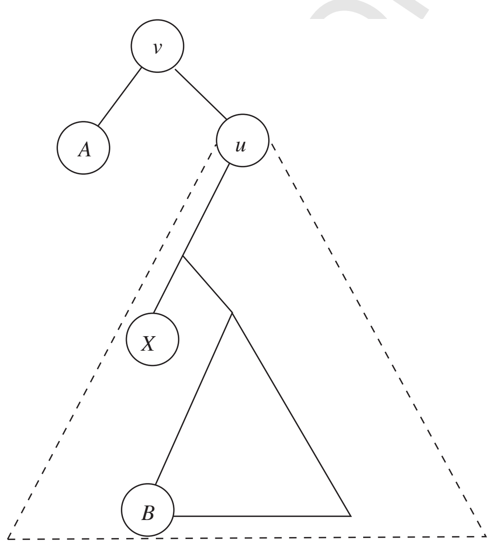

Fig. 9. Departure of A may cause a problem, if v is not in the last level of some node B in its subtree. To overcome, A and X are swaped where X is the left-most of a node that does not have this problem, then the new A is removed.

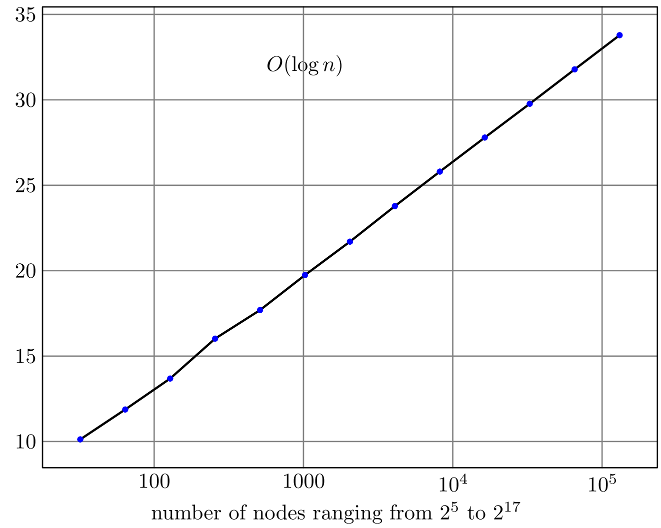

Fig. 10. Average number of links per node for networks of different sizes.

From Theorem 2, it is implied that the planes are distributed almost uniformly in each direction which is an advantage over the default case where the plane equations are chosen randomly.

7. Simulation results

For experimentation, we used a simulated case with 2^5 to 2^{17} nodes. The network started with one node and extended as the new nodes were inserted. We used a 64-dimensional vector of real numbers as our data space (in reality the fields of a database record could be of some other type like integer, string, etc. however as long as we can define a total ordering on the values of a field we do not care about its actual type, so we have used real numbers for all the 64 fields.) We note that the complexity and the depth of the query propagation is not dependent on the dimension of the data, neither is it dependent on the number of records (points) in our database. They will however affect the performance of the system but that is only proportional to the size of the result of the query.

As shown in Fig. 10, the number of links for each node is logarithmic in the size of the network. Also, Fig. 11 shows the maximum size of CPE for each node which is clearly $O(\log n)$.

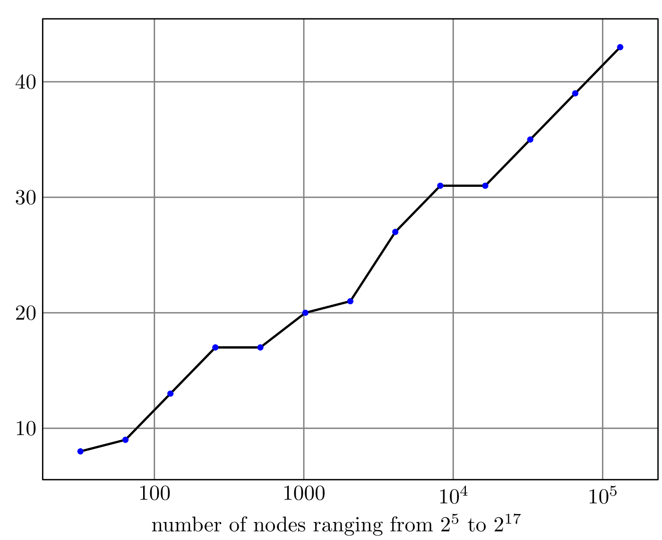

Fig. 11. Maximum CPE size per node for networks of different sizes.

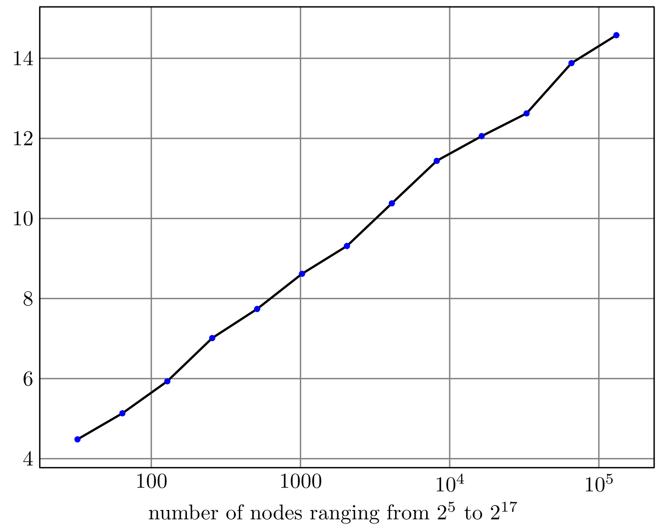

Fig. 12. Average depths of point queries.

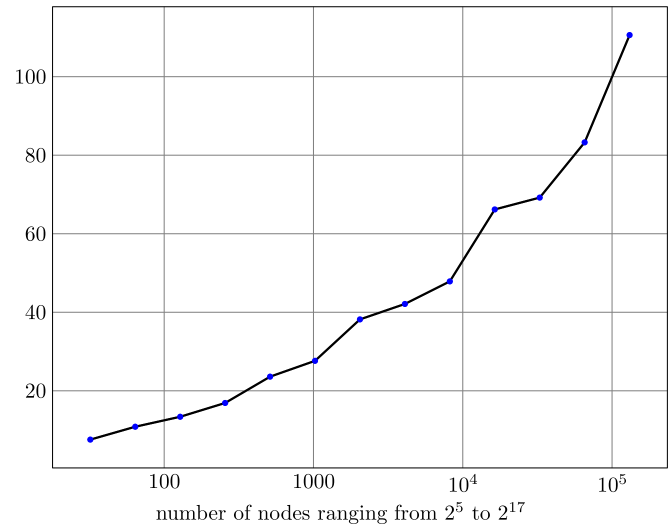

Fig. 14. Average number of nodes involved in range queries.

Figs. 12 and 13 respectively show the average depths of point and range queries which match our claims. For range queries, we have used ranges of approximately one third of the dimension size that were randomly spread throughout each dimension. This size and the randomness were the same for different network sizes. Also the total space size for different networks was assumed to be equal. Therefore, the number of nodes involved in a range query is increased for larger network sizes as shown in Fig. 14.

In another experimentation, we chose RAQ [19], a multidimensional non-DHT based P2P structure that efficiently supports point and range queries which is comparable to SkipTree. There are many DHT-based P2P systems that do not support range queries, and we thus found not suitable to compare. We compared the average number of hops needed for point and range queries of SkipTree and RAQ for different network sizes. The results are shown in Figs. 15 and 16 in logscale plots. Although these two systems perform logarithmically in terms of network size, as claimed by both works, we found out SkipTree is much faster as shown in the figures.

8. Conclusion and future work

In this paper we introduced *SkipTree* which is designed to handle point and range queries over a multidimensional space in a distributed environment. Our data structure maintains $O(\log n)$ links for each node and guarantees an upper bound of $O(\log n)$ messages w.h.p for point queries and also guarantees range queries with depth of $O(\log n)$ message w.h.p. It improves the previously proposed data structures for multidimensional space which were based on binary trees in the following aspects:

- *Links:* every node in a SkipTree keeps track of $O(\log n)$ links regardless of the shape of the tree in contrast to other tree based structures where each node should keep track of $O(h)$ links, where h (the height of the tree) can be of $O(n)$ for an unbalanced tree.
- *Query depth:* the maximum depth of a point and range query in a SkipTree is of $O(\log n)$ regardless of the shape of the tree, in contrast to other tree based structures where a query may travel $O(h)$ hops in the worst case where h can be of $O(n)$ for an unbalanced tree.

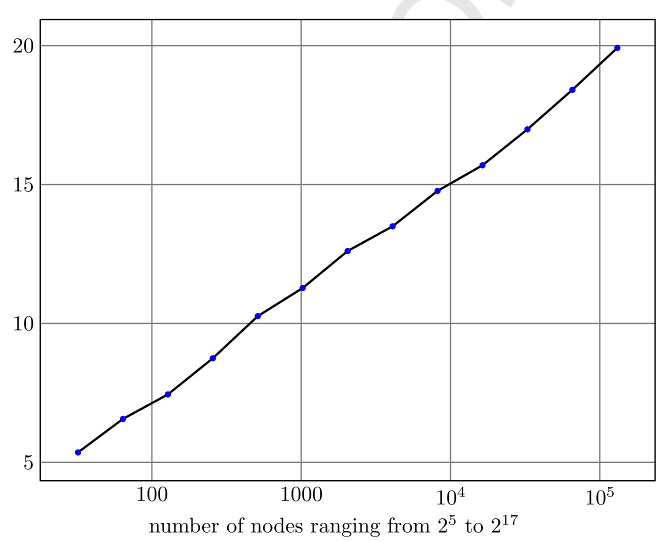

Fig. 13. Average depths of range queries.

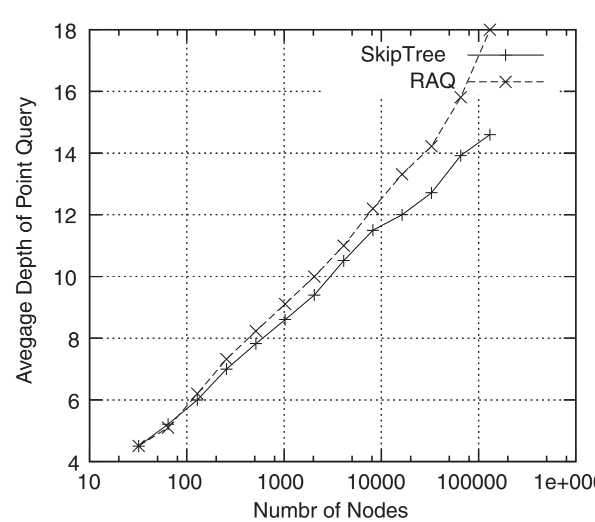

Fig. 15. Comparison between performance of SkipTree and RAQ for point query.

Please cite this article in press as: S. Alaei et al., Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries, Comput. Commun. (2009), doi:10.1016/j.comcom.2009.08.001

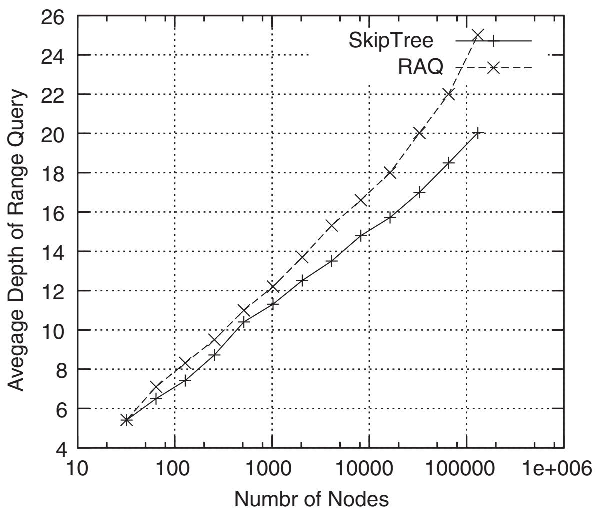

Fig. 16. Comparison between performances of SkipTree and RAQ for range query.

– *Memory requirement:* using the memory optimization of Section 6, each node needs only to store the RCPE of itself and its links that requires $O(\log h \log n)$ which is quite an improvement over memory requirement of similar tree-based structures in which each node maintains information for every node along its principal path which requires $O(h)$ memory that can be as bad as $O(n)$ for unbalanced trees.

In addition to the above improvements, we also adapted some load balancing techniques to improve our data structure. However it seems that the load balancing procedure and the memory optimization technique may be conflicting. In fact in some situation the node swapping method described in Section 6.1 may cancel out the effect of the load balancing method. This is one important area which needs further investigation. Another important area which needs further improvement is on the fault tolerance of the structure in presence of node failures. Also, as mentioned above load balancing and memory optimization need more improvements.

We presented some experimental results that verify our theoretical proofs. We also compared the performances of SkipTree and RAQ for point and range queries.

Acknowledgements

The authors would like to thank anonymous referees for their fruitful comments and thank Javad Shahparian for his help is preparing the second experimental results on comparison of SkipTree and RAQ.

References

[1] K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, P-grid: a self-organizing structured P2P system, SIGMOD Record 32 (3) (2003) 29–33.

[2] K. Aberer, M. Punceva, M. Hauswirth, R. Schmidt, Improving data access in P2P systems, IEEE Internet Computing 6 (1) (2002) 58–67.

[3] K. Aberer. Scalable data access in P2P systems using unbalanced search trees, in: Proceedings of Workshop on Distributed Data and Structures (WDAS-2002), 2002.

[4] S. Alaei, M. Toosi, M. Ghodsi. Skiptree: a scalable range-queryable distributed data structure for multidimensional data, in: 16th Annual International Symposium on Algorithms and Computation (ISAAC'2005), 2005, pp. 298–307.

[5] A. Crainiceanu, P. Linga, J. Gehrke, J. Shanmugasundaram, Querying peer-to-peer networks using p-trees, in: WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases, ACM Press, 2004, pp. 25–30.

[6] P. Fraigniaud, P. Gauron, D2b: A de Bruijn based content-addressable network, Theoretical Computer Science 355 (1) (2006) 65–79.

[7] D. Guo, J. Wu, H. Chen, X. Luo Moore, An extendable peer-to-peer network based on incomplete Kautz digraph with constant degree, in: 26th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM, IEEE, 2007, pp. 821–829.

[8] N. Harvey, M.B. Jones, S. Saroiu, M. Theimer, A. Wolman. Skipnet: a scalable overlay network with practical locality properties, in: Proceedings 4th USENIX Symposium on Internet Technologies and Systems (USITS '03), 2003.

[9] T. Johnson, A. Colbrook, A distributed data-balanced dictionary based on the b-link tree, Proceedings of the 6th International Parallel Processing Symposium, IEEE Computer Society, 1992, pp. 319–324.

[10] M. Kaashoek, D. Karger, Koorde: a simple degree-optimal distributed hash table, in: Proceedings of 2nd IPTPS, 2003.

[11] D.R. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, in: SPAA '04: Proceedings of the Sixteenth Annual ACM symposium on Parallelism in Algorithms and Architectures, ACM Press, 2004, pp. 36–43.

[12] B. Kroll, P. Widmayer, Distributing a search tree among a growing number of processors, in: SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, ACM Press, 1994, pp. 265–276.

[13] B. Kroll, P. Widmayer, Balanced distributed search tree do not exit, in: WADS '95: Proceedings of the 4th International Workshop on Algorithms and Data Structures, Springer-Verlag, 1995, pp. 50–61.

[14] D. Liben-Nowell, H. Balakrishnan, D. Karger, Analysis of the evolution of peer-to-peer systems, in: PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, ACM Press, 2002, pp. 233–242.

[15] W. Litwin, M.-A. Neimat, D.A. Schneider, Rp*: a family of order preserving scalable distributed data structures, in: J.B. Bocca, M. Jarke, C. Zaniolo (Eds.), VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 1994.

[16] W. Litwin, M.-A. Neimat, D.A. Schneider, Lh* – scalable a distributed data structure, ACM Transactions on Database Systems 21 (4) (1996) 480–525.

[17] D. Li, X. Lu, J. Wu, FissionE: a scalable constant degree and low congestion dht scheme based on Kautz graphs, in: 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005), IEEE, 2005, pp. 1677–1688.

[18] D. Malkhi, M. Naor, D. Ratajczak, Viceroy: a scalable and dynamic emulation of the butterfly, in: PODC '02: Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing, ACM Press, 2002, pp. 183–192.

[19] H. Nazerzadeh, M. Ghodsi, RAQ: a range-queriable distributed data structure (extended version), Proceeding of Sofsem 2005, 31st Annual Conference on Current Trends in Theory and Practice of Informatics, LNCS 3381, 2005, pp. 264–272.

[20] W.S. Ng. Peerdb: A P2P-based system for distributed data sharing, in: Intl. Conf. on Data Engineering (ICDE), 2003.

[21] A.M. Ouksel, G. Moro. G-grid: a class of scalable and self-organizing data structures for multi-dimensional querying and content routing in P2P networks, in: Lecture Notes on Computer Science, vol. 2872/2005, 2005, pp. 123–137.

[22] A.I.T. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Springer-Verlag, Heidelberg, 2001, pp. 329–350.

[23] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2001, pp. 149–160.

[24] K.C. Zatloukal, N.J.A. Harvey, Family trees: an ordered dictionary with optimal congestion, locality, degree, and search time, SODA '04: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2004, pp. 308–317.

[25] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz. Tapestry: a global-scale overlay for rapid service deployment, IEEE Journal on Selected Areas in Communications, 2003. Special Issue on Service Overlay Networks, in press.. Q2