SkipTree: A Scalable Range-Queryable Distributed Data Structure for Multidimensional Data

Saeed Alaei, Mohammad Toossi, and Mohammad Ghodsi

Computer Engineering Department,
Sharif University of Technology, Tehran, Iran

Abstract. This paper presents the SkipTree, a new balanced, distributed data structure for storing data with multidimensional keys in a peer-to-peer network. The SkipTree supports range queries as well as single point queries which are routed in $O(\log n)$ hops. SkipTree is fully decentralized with each node being connected to $O(\log n)$ other nodes. The memory usage for maintaining the links at each node is $O(\log n \log\log n)$ on average and $O(\log^2 n)$ in the worst case. Load balance is also guaranteed to be within a constant factor.

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* [9] refers to this class of data structures. Litwin et al. modified the original hash-based LH* structure to support range queries in RP*[8, 9]. Based on the previous work of distributed data structures, RP* and Distributed Random Tree (DRT) [6], new data structures based on either hashing or key comparison have been proposed like Chord[13], Viceroy[10], Koorde[4], Pastry[12], and P-Grid [2]. Most existing peer-to-peer overlays require $\Theta(\log n)$ links per node in order to achieve $O(\log n)$ hops for routing. Viceroy and Koorde 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.

Typically, those systems which are based on DHTs and hashing lack range-query, locality properties and control over distribution of keys due to hashing. In contrast, those which are based on key comparison, although requiring more complicated load balancing techniques, do better in those respects. P-Grid 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. Other systems like P-Tree have been proposed that provide range queries in single dimensional space.

SkipNet [3] on which our new system relies heavily, is another system for single dimensional spaces based on an extension to skip lists which supports range queries in single dimensional case too.

RAQ [11] is also another solution for the multidimensional case which incorporates a distributed partition tree structure to partition the space. Its network 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 [1] that even for such unbalanced trees the number of messages required to resolve a query still remains of $O(\log n)$ on average if the links are chosen randomly, the number of links that 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 storage of keys in multidimensional spaces. Our system uses a distributed partition tree to partition the space into smaller regions with each network node being a leaf node of that tree and responsible for one of the regions. 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 independent of the shape of the partition tree, so in general our system does not need to balance the partition tree. It 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 quiet different due to the multidimensional nature of the SkipTree. From another point of view, our system can be seen as an extension to the SkipNet for the multidimensional spaces.

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 that needs to be stored in each node. In section 3, single and range queries are explained. In section 4, the procedure for joining and leaving the network is described. In section 5, some techniques for load balancing in SkipTrees are discussed. In section 6 we modify the SkipTree structure to reduce the amount of information that needs to be stored in each node about the partition tree and finally section 7 concludes the paper.

2 Basic SkipTree Structure

The distributed data structure used in the SkipTree consists of a *Partition Tree* whose leaves also form a SkipNet.

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. v is the node responsible for every data element whose key is in $S(v)$. We extend the definition of $S(v)$ to also denote the region assigned to the internal nodes of the tree. A sample partition-tree is depicted in Figure 1.

For network node u, which corresponds to a leaf in the partition tree, we call the path connecting the root of the tree to u the *Principal Path* of node u. We

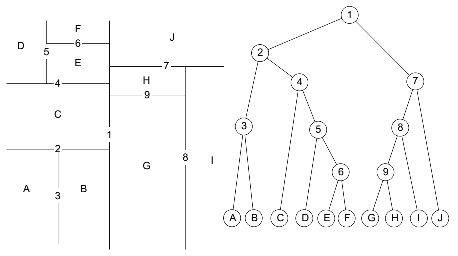

Fig. 1. A sample two dimensional partition tree and its corresponding space partitioning. Each internal node in the partition tree, labelled with a number, divides a region using the line labelled with the same number. Each leaf of the partition tree is a network node responsible for the region labelled with the same letter.

refer to the hyperplane equations assigned to the nodes of the principal path of node u (including information about on which side of those hyperplanes u resides) as the *Characteristic Plane Equations* of u or *CPE* of u for short. Every leaf node in the SkipTree stores its own CPE as well as the CPE of its links. Using theses CPE information, every node like u can locally identify if a given point belongs to a node to the left or the right of u or to the left or right of any of its links in the partition tree. The latter is useful in routing queries as explained in section 3. Hereafter, whenever we refer to a plane, we actually refer to a hyperplane of $k-1$ dimensions in a k-dimensional space.

We link the network nodes in the SkipTree together as shown in Figure 2 by forming a SkipNet among the leaves of the partition tree described before.

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 v's left and right leaf in the tree. This number is used in subsection 3.2 to handle range queries more efficiently.

3 Handling Queries

3.1 Single Point Query

The routing algorithm for single point queries is essentially the same algorithm used in the SkipNet, that is every node receiving the query along the path, sends it through its farthest link which does not point past the destination node as shown in Figure 3. The distance 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 links in the network, it guarantees routing in $O(\log n)$ hops w.h.p. A formal proof of this can be found in [3].

For the above procedure to be effective we must be able to compare points against nodes to identify whether the node containing a given point lies before or after another node in the sequence. To do so, a node compares the point against the planes in its own CPE in the order they appear in its principal path starting

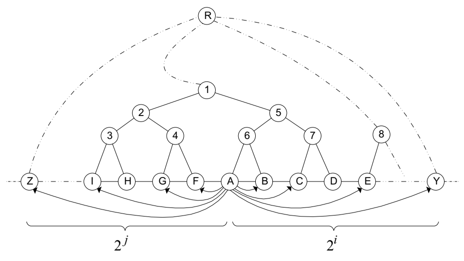

Fig. 2. The links maintained by node A in the ideal SkipTree. The target nodes are independent of the tree structure. The tree only helps us to put an ordering on the nodes. The i^{th} link in each direction skips over $2^{i-1} - 1$ nodes in that direction.

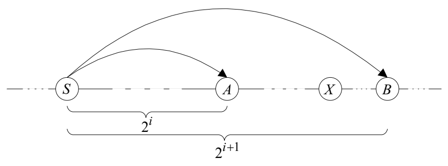

Fig. 3. A point query is routed through the farthest link which does not point past the destination node. Here, S receives a query targeting node X, so it routes the query to A. The distance to the destination node is at least halved at each hop.

from the root until it finds the first plane where the current node and the point lie on different sides of the plane. This is where the point is contained in a region belonging to a sibling subtree. If that subtree is a left (right) subtree, all of its nodes as well as the node containing the point must also be to the left (right) of the current node. The above procedure leads to $O(min(h \log n, n))$ memory usage at each node for storing the CPE. We will modify the tree structure in section 6 to improve this.

3.2 Range Query

A range query in the SkipTree is a 3-tuple of the form (R, fs, ls) where R is the query range in the multidimensional space and only the network nodes whose sequence numbers reside in the interval $[fs, ls]$ are searched. A normal range query takes the form $(R, -\infty, +\infty)$, so that all of the nodes are searched regardless of their sequence numbers. Note that the region defined by R can be of any shape as long as every node can locally identify whether R intersects with a given hypercube.

When a node S receives a range query (R, fs, ls) it sends the query to each of its links whenever there is node which intersects with the region R between that link and the next link. Assume that A and B shown in Figure 4 are two nodes corresponding to some two consecutive links maintained by S. S sends a copy of the query to A if there is any node between A and B which intersects R. Every such node, if any, must reside in one of the crosshatched subtrees illustrated in

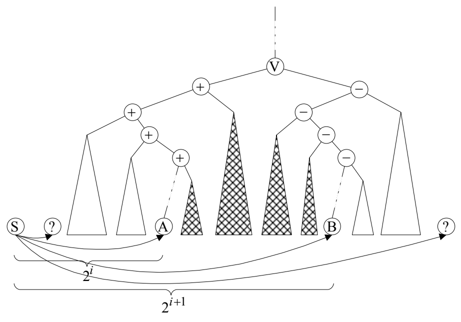

Fig. 4. A range query is propagated through each of the links maintained by S whenever there is node which intersects R between that link and the next link. Here, a copy of the query is propagated to A if any of the nodes between A and B intersects with R.

the figure. In fact, such a node must be to the right of the nodes marked with $+$ and to the left of the node marked with $-$ and because S has all of CPEs corresponding to its links, it also has access to 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 and from that it can determine whether there is any subtree between A and B whose region intersects with R and if there is such a subtree, it must also contain a node whose region intersects with R. Note that the fs and ls fields of the query are 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 Figure 4, suppose that a copy of the form (R, fs, ls) is to be sent from S to A. Also assume that $A.seq$ and $B.seq$ are the sequence number of A and B respectively. Then S computes the interval $[fs', ls']$ as the intersection of $[fs, ls]$ and $[A.seq, B.seq]$ and it sends the query (R, fs', ls') to A. This will ensure that no nodes in the network receives the query more than once.

4 Node Join and Departure

4.1 Joins

To join the SkipTree, a new node v has to be able to contact an existing node u.

The node v first inserts itself in the partition tree by splitting $S(u)$ using a new plane P to two regions. One region is then assigned to v while u retains control of the other region. Also, v copies its CPE from u and appends P to both CPEs. The plane P can be arbitrarily chosen as our load balancing protol will gradually change the partitioning to a more balanced configuration.

After updating the Partition Tree, v establishes its network links by joining the SkipNet. Node sequence numbers are used here to define a total ordering among the nodes. The SkipNet join algorithm is described in [3] and involves only $O(\log n)$ steps w.h.p.

To complete the join, u transfers the data items which are no longer in its assigned region to v.

4.2 Departures

Similar to node joins, when the node v is leaving the SkipTree, it has to follow three steps.

The first step is to update the Partition Tree. Suppose that the last plane in CPE_v, called P, splits its parent region into regions $S(v)$ and R. To update the Partition Tree, node v sends a special range query to the nodes in R and instructs them to remove the plane P from their CPE. This will effectively remove v from the partition tree.

Next step is 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 collect all possible target regions and evaluate the queries locally.

In the last step, v has to remove itself from the SkipNet. As [3] points out, this can be reduced to removing $O(1)$ links by using background repair processes similar to Chord and Pastry.

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 nonuniform key distribution.

Our load balancing protocol is derived from the *Item Balancing* technique in [5]. Load balancing is achieved using a randomized algorithm that requires a node to be able to contact random nodes in the network.

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* [7].

Load Balancing Test. In a load balancing test, node i asks a randomly chosen node j for l_j. If $l_j \geq \alpha l_i$ or $l_i \geq \alpha l_j$, 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 subsection 4.2. Then, i joins the network once again at node j and selects a hyperplane for the newly created internal node in the partition tree 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. *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 [5].

6 Memory Optimization

In this section we enforce some constraints on the plane equations that a node stores, 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:

– The planes must be perpendicular to a principal axis. So, in a k-dimensional space of $(x_1, x_2, \cdots, x_k)$ it must take the form of $x_i = c$ for some $1 \leq i \leq k$ and some value of c.
– If the search space is k-dimensional, we precisely define the form of the plane equation that may be assigned to an internal node depending on the depth of that node. We first introduce the following notation:

  d_A: for a node A in the SkipTree, the depth of A is represented by d_A and is defined to be the length of the principal path corresponding to A plus one as illustrated in Figure 5.

  l_A: for every node A in the SkipTree, the level of A is indicated by l_A where $l_A = \lceil \log_2 (\frac{d_A}{k} + 1) \rceil$ as illustrated in Figure 5.

  d'_A: for a node A in the SkipTree, the relative depth of A is represented by d'_A and is defined as $d'_A = d_A - k(2^{l_A - 1} - 1)$ as illustrated in Figure 5.

  s_A: for a node A in the SkipTree, the section number of A is represented by s_A where $s_A = \lceil \frac{d'_A}{k} \rceil$.

  We are now ready to state the last constraint:
  If A is an internal node, the plane equation assigned to A must be of the form $x_{s_A} = c$ for an arbitrary value of c, that is for any given i, all of the nodes

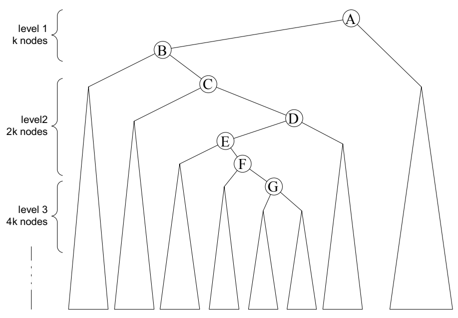

Fig. 5. A sample SkipTree for a two dimensional space. Nodes A to G have depths 1 to 7 respectively. A and B are on level 1; C, D, E and F are on level 2 and G is on level 3. The relative depth are: $d'_A = 1$, $d'_B = 2$, $d'_C = 1$, $d'_D = 2$, $d'_E = 3$, $d'_F = 4$, $d'_G = 1$.

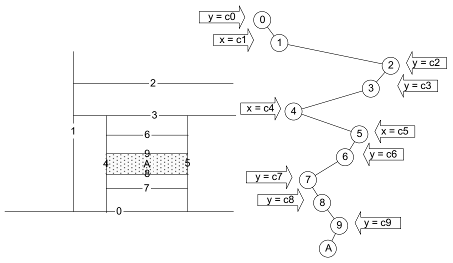

Fig. 6. The left is a sample partitioning of a 2-dimensional space under the memory optimization constraints, from the view point of node A and the right is the principal path of node A. The plane equations assigned to the internal nodes are shown in the arrows.

whose section numbers are i are assigned plane equations of the form $x_i = c$. A typical 2-dimensional space partitioned under the above constraints and its associated tree are shown in Figure 6.

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 these 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. So for each of the sections, A needs to store an inequality of the form $a \leq x_i < b$. Therefore an RCPE can be stored as an ordered sequence of inequalities of the form $a \leq x_i < b$, 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 we introduced on the beginning of this section 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 and the destination node will be 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 Figure 6 is shown bellow:

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

6.1 Node Join and Departure

Joining mechanism is the same as before except that a new node must obey the constraints mentioned earlier. However, when leaving, the situation is a little different since deleting a node may cause an internal node and its associated plane to be deleted which in turn may invalidate the memory optimization constraints. If this is the case we can swap the node to be deleted with a lower node in the tree which can be deleted without causing any problem. and then we can delete the node.

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.

7 Conclusion and Future Work

In this paper we introduced the *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 at 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. Besides, using the memory optimization of section 6, each node needs only to store the RCPE of itself and its links that requires $O(\log h \log n)$. We also adapted some load balancing techniques and a memory optimization technique to improve our data structure. Another important areas which needs further investigation is fault tolerance in presence of node failures.

References

1. K. Aberer. Scalable data access in p2p systems using unbalanced search trees. In *Proceedings of Workshop on Distributed Data and Structures(WDAS-2002)*, 2002.
2. K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3):29–33, 2003.
3. N. HARVEY, M. JONES, S. SAROIU, M. THEIMER, and A. WOLMAN. Skipnet: A scalable overlay network with practical locality properties, 2003.
4. M. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table, 2003.
5. D. R. Karger and 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*, pages 36–43. ACM Press, 2004.
6. B. Kroll and 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*, pages 265–276. ACM Press, 1994.

7. D. Liben-Nowell, H. Balakrishnan, and 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*, pages 233–242. ACM Press, 2002.
8. W. Litwin, M.-A. Neimat, and D. A. Schneider. Rp*: A family of order preserving scalable distributed data struc tures. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, sep 1994.
9. W. Litwin, M.-A. Neimat, and D. A. Schneider. Lh* – a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525, 1996.
10. D. Malkhi, M. Naor, and 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*, pages 183–192. ACM Press, 2002.
11. H. Nazerzadeh and M. Ghodsi. RAQ: A range-queriable distributed data structure (extended version). In *Proceeding of Sofsem 2005, 31st Annual Conference on Current Trends in Theory and Practice of Informatics, LNCS 3381, pp. 264-272*, February 2005.
12. A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.
13. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. pages 149–160.