Skip B-Trees

Ittai Abraham[1], James Aspnes[2*], and Jian Yuan[3]

[1] The Institute of Computer Science, The Hebrew University of Jerusalem, ittaia@cs.huji.ac.il
[2] Department of Computer Science, Yale University, aspnes@cs.yale.edu
[3] Google, yuanjian@gmail.com

Abstract. We describe a new data structure, the **Skip B-Tree**, that combines the advantages of skip graphs with features of traditional B-trees. A skip B-Tree provides efficient search, insertion and deletion operations. The data structure is highly fault tolerant even to adversarial failures, and allows for particularly simple repair mechanisms. Related resource keys are kept in blocks near each other enabling efficient range queries.

Using this data structure, we describe a new distributed peer-to-peer network, the **Distributed Skip B-Tree**. Given m data items stored in a system with n nodes, the network allows to perform a *range search* operation for r consecutive keys that costs only $O(\log_b m + r/b)$ where $b = \Theta(m/n)$. In addition, our distributed Skip B-tree search network has provable polylogarithmic costs for all its other basic operations like insert, delete, and node join. To the best of our knowledge, all previous distributed search networks either provide a range search operation whose cost is worse than ours or may require a linear cost for some basic operation like insert, delete, and node join.

1 Introduction

Peer-to-peer systems provide a decentralized way to share resources among machines. An ideal peer-to-peer network should have such properties as decentralization, scalability, fault-tolerance, self-stabilization, load-balancing, dynamic addition and deletion of nodes, efficient query searching and exploiting spatial as well as temporal locality in searches.

Much of academic work on peer-to-peer systems has concentrated on building **distributed hash tables** or DHTs. In a DHT, the hash value of the key of resource is used to determine which node it will be stored at (typically the node whose own hashed identity is closest), and the use of random-looking hash values roughly balances out the load on the nodes in the system. An overlay graph is then constructed on top of the nodes in order to allow efficient searches for the nearest node to a target hash using some sort of routing algorithm. The major form of variation between these DHTs is the routing algorithm used to locate

* Supported in part by NSF grants CCR-0098078, CNS-0305258, and CNS-0435201.

resources; however, in each case the underlying structure is built on pointers between nodes, so the resulting mechanism typically looks like some sort of tree search.

Even though traditional DHT systems effectively construct balanced search trees in order to find nodes, they generally do not support range queries since hashing destroys the ordering on keys. They also typically lack load balancing mechanisms other than the limited randomized balancing provided by hashing. For example, in Chord it is likely that some machine will own $\Omega(\log N/N \log\log N)$ fraction of the key space. There are some recent extensions of DHT systems which try to mitigate this problem. An extension of Chord called a p-tree [CLGS04] supports $O(\log_b N)$ search as well as providing efficient range query. However, there is no analysis on deletion and insertion, and the addition and removal of nodes are based on a complicated self-stabilization mechanism whose performance is based on empirical data only. Karger and Ruhl [KR04] propose algorithms to do address space balancing and item balancing in Chord, which ensures with high probability no node will be responsible for more than $O(1/N)$ of the key space. The item balancing algorithm is dependent on nodes being able to move freely in the key space and is incompatible with the address space balancing algorithm though. Ratnasamy etc. [RRHS04] proposes a new data structure called Prefix Hash Tree (PHT) that could be put on top of existing DHT. PHT is essentially a binary trie built over data sets being indexed. The system supports range queries and is load balanced, but it suffers from hot spots since the top-level trie nodes tend to be accessed more frequently than bottom-level trie nodes.

Though continued research on DHTs is likely to lead to further improvements, some of the difficulties with reconciling range queries and DHT structures is inherent in the use of hashing to perform load balancing. Another line of research has focused on providing searchable concurrent data structures by applying the tree structure in order to support efficient range queries using mechanisms similar to those in traditional balanced binary trees. For example, **Skipnet**, developed by Harvey etc. [HJS+03], is a trie of circular, singly-linked skip lists that link the machines in the system. It provides path locality and content locality, and its hashing provides some form of load balancing. However, transparent remapping of resources to other domains is not possible. Aspnes and Shah [AS02] concurrently devised a data structure called a **skip graph** which applies skip lists in a similar way to support $O(\log N)$ search, insertion and deletion operations, while maintaining the inherent tree structure in the network so that range queries are also supported. Skip graphs are also tolerant to node failures, including both adversarial failures and random failures.

The original skip graph construction in [AS02] was marred by the lack of any policy for assigning resources to nodes, excessive internode pointers, and a cumbersome self-repair mechanism. Recently Aspnes *et al.*[AKK04] have proposed a mechanism to do global load balancing by pairing heavily loaded machines with lightly loaded ones, while using sampling to reduce the number of pointers in the data structure from $O(\log N)$ per resource to $O(\log N)$ per machine. However,

search times in such binned skip graphs still suffer from large constants, and exploiting the large memory capacity of typical machines may allow much faster searching.

1.1 Our contribution

We describe a new data structure, the **Skip B-Tree**, which has the following features:

1. By combining skip graphs with features of traditional B-trees, the skip B-Tree avoids the drawbacks of traditional skip graphs while providing $O(\log_b N)$ search, insertion and deletion operations, where b is the block size. When $b = N^{1/k}$ for some constant k, then for any set of N items, all operations take constant time, $O(k)$.
2. The high connectivity of our data structure makes it highly fault tolerant even to adversarial failures, and allows for particularly simple repair mechanisms.
3. Related resource keys are kept in blocks near each other, which may enhance the performance of applications such as web page prefetching which utilize the locality of resources.

Using this data structure, we describe a new distributed network, the **Distributed Skip B-Tree**. We show that our distributed Skip B-tree is the first distributed search network with provable polylogarithmic costs for all its basic operations[4]. It employs balancing techniques from [AAA+03] to locally update system parameters and hence avoids costly global re-balancing. Moreover, given m data items stored in a system with n nodes, a range search for r consecutive keys costs only $O(\log_b m + r/b)$ where $b = \Theta(m/n)$. To the best of our knowledge, all previous distributed search networks may require a linear cost for some operation or do not provide cost efficient range queries. Aspnes *et al.*[AKK04] has a load balancing scheme that may cause an insert operation to trigger a global re-balancing that costs $\Omega(n)$. Awerbuch and Scheideler [AS03] have a scheme for which a range search for r consecutive keys costs $O(r \log n)$. Hence their solution obtains no locality of resources and incurs a high cost relative to our solution.

1.2 Distributed Search Trees vs Distributed Hash Tables

Skip B-trees are instances of the general concept of **Distributed Search Trees** (DSTs), which we now define. Essentially, DSTs are to search trees what DHTs are to hash tables. We begin by defining the interface to a **Distributed Hash Table** (DHT). A DHT is a distributed network on n nodes storing m (key,value) pairs with the following operations.

1. *Add*: Add a node to the system.

[4] See Section 1.2 for a formal definition of the operations and their cost measures

2. *Remove*: Gracefully remove a node from the system.
3. *Insert*: add a (key,value) pair.
4. *Delete*: remove a (key, value) pair.
5. *Search*: Given a key, find the corresponding value(s).

The typical cost measures of a DHT are to achieve worst case guarantees for the following:

1. Network change cost: Message complexity of Add or Remove operations. For example $O(\log^2 n)$ in Chord [SMLN+03] and $O(\log n/\sqrt{\log\log n})$ in [KM05].
2. Data change cost: Message complexity of Insert, Delete, and Search operations. For example, $O(\log n)$ in [SMLN+03] and $O(\log n/\log\log n)$ in Koorde [KK03].
3. Data load: The maximal fraction of data items stored in one machine. For example $O(\log n/n)$ in [SMLN+03] and $O(1/n)$ in [KR04].
4. Network load: The maximal fraction of traffic a node receives given that random nodes search for random data. For example $O(\log n/n)$ in [SMLN+03].

The interface of a DST contains all the operations of a DHT and includes one new operation, the *Range search*. This search operation gets two parameters (k, r) and must return the r minimal keys whose value is larger than the search key k (one can also require a search for the r keys that are smaller than k). The cost metrics for DSTs are the same as for DHTs with the only difference being that the complexity of a range search operation is measured as a function of the required range r. Ideally, an efficient distributed search tree that stores m data items over a network with n nodes should store the index sorted with each node storing a consecutive block of size $b = \Theta(m/n)$ of the index. In such a case a range search operation for r keys should ideally require only $O(\log_b m + r/b)$ messages. Indeed we will show that our solution obtains this asymptotic bound while keeping all other operations at a polylogarithmic cost.

Finally we mention that handling faulty nodes (non-graceful node removals) is also an important issue both for DHTs and for DSTs. This usually requires data replication and techniques that are out of the scope of this short paper.

2 Skip B-trees

The B-tree was originally introduced by Bayer [Bay72]. The B-tree algorithms utilized the locality of data and were designed to minimize the cost of sequential search/insert/delete operations. There has been a lot of research on building a distributed B-tree that supports concurrency and parallelism. Gilon and Peleg [GP91] proposed several structures for implementing a distributed dictionary, with the focus on reducing complexity of message passing as well as data balancing. Colbrook etc. [CBDW91] have proposed a pipelined distributed B-tree. Johnson etc. [JC94] describe a data structure called a dB-tree which permits concurrent updates on a replicated tree node, and rarely blocks operations.

A skip graph, introduced by Aspnes and Shah [AS02], is organized as a tower of increasing sparse linked lists, much like a skip list [Pug90]. Level 0 of a skip graph is just a doubly linked list of all nodes in increasing order by key. For each i greater than 0, each node appears randomly in one of the many link lists in level i (unlike a skip list where there is only one linked list per level), with two constraints. First, if node x is a singleton at level $i-1$, it doesn't appear in any of the linked list at levels higher than $i-1$. Second, for every linked list L at level i, there must be another linked list L' at level $i-1$ where the elements in L are a subset of the elements in L'.

Our skip B-tree can be viewed as a non-trivial extension of the skip graph, combined with the idea of a distributed B-tree. We specify a block size b, and for every linked list on any level we divide it into blocks where the expected size of each block is $O(b)$ (we will explain how to do this later). The division into blocks is independent of the skip graph structure.

As in a skip graph, each element x is assigned a membership vector $m(x)$, where the characters in $m(x)$ are taken from a finite alphabet set Σ. The cardinality of the alphabet, $|\Sigma|$, is typically taken to be the same as the block size b. Every doubly-linked list in the skip B-tree is labeled by some finite word w. An element x is in the list labeled by w if and only if w is a prefix of $m(x)$. Each element in the block keeps two pointers, one to the corresponding element in the upper level (called "parent") and one to the corresponding element in the lower level (called "child"). The block itself keeps two pointers to its two neighbors at the same level. It also keeps a count of how many elements there are in the block. According to Lemma 1, the expected height of the skip B-tree is $O(\log_b N)$. By making b large enough (say $b = 10^7$), in practice the height of a skip B-tree can be a very small constant (say, 2 or 3 for any data set).

We adopt much of the notation of [AS02]. In particular, for any element w, write $w \upharpoonright i$ for the prefix of w of length i. Write ϵ for the empty word. For each block b at level ℓ, write $m_ell(b)$ to denote the ℓ-th character of the membership vector of any element in b. In the implementation of the algorithm we actually store a number in each block indicating the ℓ-th element in the membership vector of all elements belonging to it instead of storing a membership vector in each element.

As in a skip graph, the bottom level of a skip B-tree is always a doubly-linked list S_ϵ consisting of all the nodes in order, divided into blocks with size of $O(b)$. In general, for each w in Σ^*, the doubly-linked list S_w contains all x for which w is a prefix of $m(x)$, in increasing order, divided into blocks with size of $O(b)$. We say that a particular list S_w is part of level i if $|w| = i$. This gives an infinite family of doubly-linked lists; in an actual implementation, only those S_w with at least two nodes are represented.

Lemma 1. *With high probability, the height of a skip B-tree is* $O(\log_b N)$.

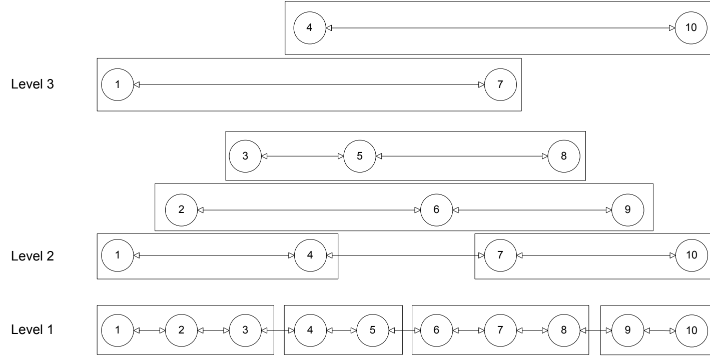

Fig. 1. A skip B-tree with $n = 10$ nodes and $\lceil \log_b n \rceil = 3$ levels. The block size $b = 3$.

3 Algorithms for a skip B-tree

Here we describe the search, insert and delete operation for a skip B-tree. We summarize the variables stored at each node in Table 1. For simplicity, our description assumes a supply of **blocks** that can hold many data items. The question of how these blocks are mapped to actual physical machines is deferred to Section 6.

Variable	Meaning
MaxKey	the maximum resource key in a block
MinKey	the minimum resource key in a block
currentBlock	the block receiving the message
Right	the right neighbor of the current block
Left	the left neighbor of the current block
Level	the level of the block
m	Membership vector
$[key]$	the element in the block indexed by key
Parent	pointer to the block one level higher which contains the same resource key as the element
Child	pointer to the block one level lower which contains the same resource key as the element
Group	indicates the grouping of the block

Table 1. List of all the variables stored at each node.

In this section, we will give the algorithms and analyze their performance.

3.1 The search operation

The search operation (Algorithm 1) is basically the same as that of a skip list, except that our unit of search is now a block. The search is initiated by a top level

block seeking a key and it proceeds down the same level without overshooting, continuing at a lower level if required, until it reaches level 0. Either the block at level 0 which contains the key, if it exists, or the block at level 0 storing the key closest to the search key is returned. The algorithm is described below:

Algorithm 1: search for the file indexed by *searchKey*

```
upon receiving ⟨searchOp, startBlock, searchKey, level⟩:
if (searchKey exists in unmarked elements of currentBlock) then
  if (level = 0) then
    send ⟨foundOp, currentBlock⟩ to startBlock
  else
    send ⟨searchOp, startBlock, searchKey, level − 1⟩ to currentBlock[searchKey].Child

if (searchKey > currentBlock.MaxKey) then
  while (level ≥ 0) do
    if (currentBlock.Right.MinKey < searchKey) then
      send ⟨searchOp, startBlock, searchKey, level⟩ to currentBlock.Right
      break
    else if (level > 0) then
      send ⟨searchOp, startBlock, searchKey, level − 1⟩ to currentBlock[currentBlock.MaxKey].Child

else
  while (level ≥ 0) do
    if (currentBlock.Left.MaxKey > searchKey) then
      send ⟨searchOp, startBlock, searchKey, level⟩ to currentBlock.Left
      break
    else if (level > 0) then
      send ⟨searchOp, startBlock, searchKey, level − 1⟩ to currentBlock[currentBlock.MinKey].Child

if (level = 0) then
  send ⟨notFoundOp, currentBlock⟩ to startBlock
```

Lemma 2. *The search operation in a skip B-tree S with N nodes and block size b takes $O(\log_b N)$ time and $O(\log_b N)$ messages with high probability.*

Skip graphs can support range queries in which one is asked to find a key within a specified range. For most of these queries, the procedure is an obvious modification of Algorithm 1 and runs in $O(\log_b N)$ time with $O(\log_b N)$ messages. For finding all nodes in an interval, we can use a modified Algorithm 1 to find the closest element to the upper (or lower) bound. We then walk from this element in level 0 list until we hit the lower (or upper) bound, and return all the elements we have encountered. If there are r elements in the interval, the running time is $O(log_b N + r)$.

3.2 The insert operation

A new element n knows some introducing block *introducer* which helps it to join the network. n inserts itself in one list at each level until it finds itself a singleton list at top level. At level 0, n will be added to the block which contains a key closest to n.Key. At each level i, $i >= 1$, n will try to find the closest element x in level $i-1$ with $x \upharpoonright i = n \upharpoonright i$ and add to the block x belongs to at level i. To ensure load balancing, we adopt the approach described in [AKK04]. Specifically, we call a block "closed" if it has more than $b/2$ elements, and we call

it "open" if it has no more than $b/2$ elements. We group the blocks into groups of 2 or 3, with each group having the following property: it must either contain one closed block followed by one open block, or it may contain 2 closed blocks and 1 open block while the open block is in the middle. This is the invariant we try to keep for our insertion and deletion algorithm. When we insert a new element, if we insert it into a closed block we always move the largest element to the adjacent open block in the same group. If the open block is still open after insertion, nothing happens. If it is in a group of 2 and it becomes closed, we add a new empty block in the middle of these two blocks and mark it "open". We move the element to this new block instead. If the open block is in a group of 3, we create a new block, link it to the right of the rightmost closed block, and move the largest element in the open block in the middle to its neighbor to the right, which in turn causes the movement of the largest element in the rightest closed block to the new block. We also split it into two groups of size 2 since we have 4 blocks now. Notice that in this way we guarantee that the average block size of any group is no smaller than $b/4$. To simplify analysis, we do not allow duplicates here, but it is quite easy to extend the algorithm so that duplicates are allowed. Also when we create a new block, we assume that there exists a routine which allocates the space for the block and distribute it to a random machine in the network.

Algorithm 2: insert a new element n

```
if (introducer = ⊥) then
    create a new block and add n to the block
    block.Left ← ⊥
    block.Right ← ⊥
    block.Group ← 2
else
    send ⟨searchOp, currentBlock, n.Key, introducer.Level⟩ to introducer
    wait until foundOp or notFoundOp is received
    upon receiving ⟨foundOp, clone⟩:
        terminate insert
    upon receiving ⟨notFoundOp, block⟩:
    childblock ← ⊥
    while true do
        level ← block.Level
        send ⟨buddyOp, currentBlock, n, n.level, ⊥⟩ to block
        wait until receipt of ⟨setLinkOp, newblock⟩:
        send ⟨linkOp, n, childblock, newblock⟩ to block
        if (newblock ≠ ⊥) then
            childblock ← block
            block ← newblock
        else
            newBlock ← create a new block
            m(newBlock) ← uniformly chosen random element of Σ
            add n to newBlock
            n.Child ← block
            n.Parent ← ⊥
            break
```

3.3 The delete operation

Deletion works as follows: we recursively delete the element from each level it belongs to in a bottom-up fashion. When we delete an element, we check

Algorithm 3: block's message handler for physically inserting new element n.

```
upon receiving ⟨linkOp, n, childBlock, parentBlock⟩:
add n to currentBlock
n.Child ← childBlock
n.Parent ← parentBlock
currentBlock.Count++
//the block is open
if (currentBlock.Count <= b/2) then
    return
//this is the first block on this level and it is closed
if (currentBlock.Left = currentBlock.Right = ⊥) then
    block ← create a new block
    block.Group ← 2
    insert block to the right of currentBlock
    m ← largest element in currentBlock
    send ⟨linkOp, m, m.Child, m.Parent⟩ to block
    remove m from currentBlock
    return
//if the block was closed before, swap element with the open block in the group
if (currentBlock.Count > b/2 + 1) then
    if (currentBlock.Left.Count <= b/2 and currentBlock.Group = 3) then
        block ← create a new block
        block.Group ← 2
        insert block to the right of currentBlock
        currentBlock.Left.Left.Group ← 2
        currentBlock.Left.Group ← 2
        currentBlock.Group ← 2
        m ← largest element in currentBlock
        send ⟨linkOp, m, m.Child, m.Parent⟩ to currentBlock.Right
        remove m from currentBlock
        return
    else
        m ← largest element in currentBlock
        send ⟨linkOp, m, m.Child, m.Parent⟩ to currentBlock.Right
        remove m from currentBlock
    return
//currentBlock must have b/2 + 1 elements now
if (currentBlock.Group = 2) then
    currentBlock.Left.Group ← 3
    currentBlock.Group ← 3
    block ← create a new block
    block.Group ← 3
    insert block to the left of currentBlock
    m ← smallest element in currentBlock
    send ⟨linkOp, m, m.Child, m.Parent⟩ to currentBlock.Right
    remove m from currentBlock
    return
else
    m ← largest element in currentBlock
    send ⟨linkOp, m, m.Child, m.Parent⟩ to currentBlock.Right
    remove m from currentBlock
    return
```

Algorithm 4: block's message handler for finding the closest block one level higher to insert new element n, whose $b.Level$-th component of membership vector is val.

```
upon receiving ⟨buddyOp, startBlock, n, val, side⟩:
foreach (element x in currentBlock)
if (m(x.Parent) = val) then
    send ⟨setLinkOp, x.Parent⟩ to startBlock
    return
if (side = ⊥) then
    if (currentBlock.Left ≠ ⊥) then
        send ⟨buddyOp, startBlock, n, val, Left⟩ to currentBlock.Left
    if (currentBlock.Right ≠ ⊥) then
        send ⟨buddyOp, startBlock, n, val, Right⟩ to currentBlock.Right
    if (currentBlock.Left = ⊥ and currentBlock.Right = ⊥) then
        send ⟨setLinkOp, ⊥⟩ to startBlock

else
    if (currentBlock.side ≠ ⊥) then
        send ⟨buddyOp, startBlock, val, side⟩
        to currentBlock.side
    else
        send ⟨setLinkOp, ⊥⟩ to startBlock
```

the block's size. If it remains closed/open after deletion, we simply remove the element from it. Notice that we allow an empty block to be in the group here. If it changes from closed to open and the open block in the group is not empty, we move the largest/smallest in the open block to the current block. If the open block in the group is empty, we then check the group size. If it is a group of size 3, we simply remove the empty block in the middle and form a group of size 2 since the block is open now. If it is a group of size 2, we check the size of the group to the left. If it is also a group of size 2, we move the largest key in the open block of the left neighbor to the current block, delete the empty open block and form a group of 3. If it is a group of size 3, we delete the empty block, and form 2 groups of size 2 with the left neighbor. Notice that the invariant of group structure is still preserved by our deletion algorithm.

The proof of the correctness of this mechanism is essentially the same as the proof of Theorem 4 in [AKK04].

Lemma 3. *The insertion and deletion operations in a skip B-tree S with N nodes and block size b take $O(\log_b N)$ messages and $O(\log_b N)$ time with high probability.*

3.4 Concurrency issues

In order to ensure the correctness of the algorithm under concurrent updates, we need a lock-free doubly linked list in a distributed setting. Shasha and Goodman [SG88] provide a framework for proving the correctness of non-replicated concurrent data structures. For example, we could use the underlying doubly linked list of dB-tree [JC94] as our doubly linked list. Since our insertion and deletion operations all work in a bottom-up fashion, as long as each level is consistent the whole data structure must be intact, and a lock-free doubly-linked list ensures the consistency of each level. The only thing that could be missing during updates is the pointers between different levels, but this will only slow down the search operation and has no effect on the consistency of the data structure.

4 Fault tolerance

In this section, we describe some of the fault tolerance properties of a skip B-tree. Fault tolerance of related data structures, such as augmented versions of linked lists and binary trees, has been well-studied and some results can be seen in [MP84,AB96]. Section 5 gives a repair mechanism that detects node failures and initiates actions to repair these failures. Before we explain the repair mechanism, we are interested in the number of blocks that can be separated from the primary component by the failure of other blocks, as this determines the size of the surviving skip B-tree after the repair mechanism finishes.

Notice that if multiple blocks are stored on a single machine, if that machine crashes all of its blocks are lost. Our results are stated in terms of the fraction

of blocks that are lost; if the blocks are roughly balanced across machines, this will be proportional to the fraction of machine failures. Nonetheless, it would be useful to have a better understanding of fault tolerance when the mapping of resources to machines is taken into account; this may in fact dramatically improve fault tolerance, as blocks stored on surviving machines can always find other blocks stored on the same machine, and so need not be lost even if all of their neighbors in the skip B-tree are lost.

We give analysis of adversarial failures here, as this will be the worst case failure pattern. In this section we look at the expansion ratio of a skip B-tree, which gives the number of nodes that can be separated from the primary component even with adversarial failures.

Let G be a graph. Recall that the expansion ratio of a set of nodes A in G is $|\delta A|/|A|$, where $|\delta A|$ is the number of nodes that are not in A but are adjacent to some node in A. The expansion ratio of the graph G is the minimum expansion ratio for any set A, for which $1 \leq |A| \leq n/2$. The expansion ratio determines the resilience of a graph in the presence of adversarial failures, because separating a set A from the primary component requires all nodes in δA to fail. We will show that skip B-trees have $\Omega(\frac{1}{b})$ expansion ratio with high probability, implying that only $O(f \cdot b)$ nodes can be separated by f failures, even if the failures are carefully targeted.

Since all the real data is stored on level 0 blocks, we only need to consider the case when A consists entirely of level 0 blocks. The probability for a level 1 block to have no neighbor in A is $(\frac{m_0-|A|}{m_0})^b$ since none of its pointers to level 0 blocks can point to any block in A, where m_0 is the total number of blocks on level 0. Thus the expected number of neighbors at level 1 is $m_1(1-(1-\frac{|A|}{m_0})^b)$, which is greater than $\frac{b|A|m_1}{m_0}$. Since $m_1 = \Theta(m_0)$, the expansion ratio is $\Omega(\frac{1}{b})$, which is pretty good since there are only $O(|A|b)$ links from A to level 1 blocks. It is comparable to the guarantee provided by data structures based on explicit use of expanders such as censor-resistant networks [FS02,SFG+02,Dat02].

5 Repair mechanism

In this section we describe a self-stabilization mechanism that repairs our skip B-tree in case of block failure. We assume that a block either works or fails in its entirety. The repair mechanism is quite simple: each block sends message to its neighbors periodically to see if they are alive. If one of the neighbors is dead, we try to fix the link to the next live neighbor. Without loss of generality, we assume that the right neighbor fails, and the block resides on level 0.

Lemma 4. *For any two adjacent blocks b_1 and b_2 on level 0, the probability that there is an element x_1 from b_1 and an element x_2 from b_2 such that $x_1 \restriction 1 = x_2 \restriction 1$ is at least $1-e^{-b/4}$*

Algorithm 5: Algorithm for repairing right neighbor for block *block* at level 0.

```
send ⟨repairOp, block.maxKey, block⟩ to block
upon receiving ⟨repairOp, key, block⟩:
minKey ← ∞
foreach element x in block
send message to x.Parent and x.Parent.Right asking for the smallest key greater than key
if (the reply is not ⊥ and the key returned is < minKey) then
 | minKey ← the key returned
 | newBlock ← the block containing the key
 //make sure that newBlock's left neighbor is indeed missing if (newBlock.Left = ⊥) then
 | newBlock.Left ← block
 | block.Right ← newBlock
else
 | send ⟨repairOp, key, block⟩ to the left neighbor of current block
```

Thus we can see that the repair mechanism would finish in expected $O(1)$ time if we assume the node can process $O(b)$ messages simultaneously, and sends expected $O(b)$ messages with high probability.

6 Distributed Skip B-Trees

In this section, we detail how to map skip B-trees to machines and build an efficient DST. Consider a network with n machines that stores m data items, we would like to have a skip B-tree with block size $b \approx m/n$. We use the load balancing strategy of [AAA+03] in order to label nodes with $\Theta(\log n)$ identifiers. This can be done so that all nodes have unique binary identifiers that form a prefix code whose size is between $\log n - C$ and $\log n + C$ for a predetermined constant C. The add node and remove node operations maintain this invariant with cost $O(\log^2 n)$ [AAA+03].

In order to map a skip B-Tree to nodes we must map nodes to blocks in a manner that balances load between nodes and maintains low degree (an edge is formed between any two nodes that store two consecutive blocks of any of the linked lists of the skip B-Tree structure). The idea is that each node estimates b to be about m/n, the estimation of b will always be always a power of two.

We now explain how to maintain the base linked list S_ϵ that is maintained by all the network nodes. However, the same techniques are used to store all the linked lists. Specifically, for any binary word w, the nodes whose identifiers are a prefix of w maintain the linked list S_w in the same fashion.

Insertion of a block into a linked list is performed in the following manner. A sample of $\Theta(\log n)$ random nodes are queried, and the least loaded node gets to store the block. The nodes that store the previous and next blocks now store a network link to this new location and the chosen node adds links to them. If the adversary is oblivious to the random choices then with high probability [MRS01,ABKU00] all machines will have the same load (number of blocks) up to a constant factor. If $b = \Theta(m/n)$ and n nodes maintain the list then the number of blocks per node is $O(1)$ and hence the number of links of each node is also $O(1)$.

We now analyze the number of network links each node needs to maintain for all the lists it belongs to. Fix a node u with id $id(u)$, it participates in

maintaining all the linked lists S_w such that w is a prefix of $id(u)$ or $id(u)$ is a prefix of w. Hence there are $O(\log m)$ such lists. Each such list S_w with $|w| = i$ contains $\Theta(2^{-i}m)$ elements and since node identifiers are balanced there are $O(2^{-i}n)$ nodes whose prefix is a prefix of w. Since $b = \Theta(m/n)$ then each such list requires $O(1)$ links for each node maintaining it. Therefore, for maintaining all lists of the skip B-Tree the degree of each node is $O(\log m)$. So the cost of adding a node and setting up its connections is $O(\log n \log m)$.

Finally, we need a mechanism to update b as the size of n and m dynamically change over time. We want to avoid global pitfalls that would require the whole system to do a global update as such operations are not scalable. Each node maintains b at a power of two, for a node v let $b = 2^{B(v)}$ be its local estimate. Several events described below may case $B(v)$ to change. Whenever $B(v)$ changes this effects the open or closed status of the nodes blocks. We use the bucket compression technique of [AKK04] (section 3.3) and a similar bucket expansion algorithm to locally adjust the nodes blocks to the new value of $b = 2^{B(v)}$. The details will appear in the full paper.

When a node joins the system, we use the node split mechanism of [AAA+03]. When node v splits, it decreases its $B(v)$ by one and the new node also takes the updated value of $B(v)$. Similarly, when a node leaves, we use the merge mechanism of [AAA+03]. The two merged nodes decrease their $B(v)$ by one. Changes in $B(v)$ also occur due to change in the number of blocks stored. Once a node stores more that a given constant number of blocks, it locally increase its value of $B(v)$ by one. Similarly, when a node has less than a given constant number of blocks, it locally decreases $B(v)$ by one. The estimation of b is adequate since the load balancing algorithms give each node an estimate of n and m up to constant factors with high probability. In the full paper, we prove that using this strategy the nodes' estimates of b are all within a constant factor of each other. Moreover, locally updating b has low cost as only $O(\log n \log m)$ messages are sent.

One remaining obstacle is that the skip B-tree now has different members having slightly different estimates of b. As long as estimates are bounded by a constant factor it is easy to see that insert, delete, and search operations can still be carried out using $O(\log_b m)$ messages. The resulting distributed data structure is a DST with the following costs.

Lemma 5. *Given an n node network storing m items, both the network change cost and the data change cost is $O(\log n \log_b m)$. A range search for r consecutive keys costs only $O(\log_b m + r/b)$. Both the data load and network load are $O(\log n/n)$.*

7 Conclusion

In this paper we defined a new data structure called skip B-tree which has several desirable properties. Insertion, deletion and search in skip B-tree all take $O(\log_b N)$ time for any set of elements that be arbitrarily unbalanced. In practice

just as in B-trees, our cost is a very small constant (2-3) for reasonably large b (say, 10^7). Also under the condition of no additional node failures, the skip B-tree can repair itself in a very efficient way. Finally, skip B-tree also supports range queries, and it exploits the geographical proximity in location of resources. We use skip B-trees to build a distributed peer-to-peer network that provides the first polylogarithmic cost DST that allows to perform efficient range search operations.

References

AAA+03. Ittai Abraham, Baruch Awerbuch, Yossi Azar, Dahlia Malkhi, and Elan Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

AB96. Yonatan Aumann and Michael A. Bender. Fault Tolerant Data Structures. In *Proceedings of the Thirty-Seventh Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–589, Burlington, VT, USA, October 1996.

ABKU00. Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, February 2000.

AKK04. James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load balancing and locality in range-queriable data structures. In *Twenty-Third ACM Symposium on Principles of Distributed Computing*, pages 115–124, July 2004.

AS02. James Aspnes and Gauri Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms(SODA)*, pages 384–393, January 2002. Submitted to *Journal of Algorithms*.

AS03. Baruch Awerbuch and Christian Scheideler. Peer-to-peer Systems for Prefix Search. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Com puting (PODC)*, Boston, MA, USA, July 2003. To Appear.

Bay72. Rudolf Bayer. Symmetric Binary B-trees: Data Structure and Maintainance Algorithms. *Acta Informatica*, pages 290–306, 1972.

CBDW91. Adrian Colbrook, Eric A. Brewer, Chrysanthos Dellarocas, and William E. Weihl. An Algorithm for concurrent search trees. In *Proceedings of the 20th International Conference on Parallel Processing*, pages III138–III141, 1991.

CLGS04. Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. P-Tree: A P2P Index for Resource Discovery Applications. *The 13th International World Wide Web Conference*, pages 390–392, May 2004.

Dat02. Mayur Datar. Butterflies and Peer-to-Peer Networks. In Rolf Möhring and Rajeev Raman, editors, *Proceedings of the Tenth European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 310–322, Rome, Italy, September 2002.

FS02. Amos Fiat and Jared Saia. Censorship Resistant Peer-to-Peer Content Addressable Networks. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 94–103, San Francisco,

CA, USA, January 2002. Submitted to a special issue of *Journal of Algorithms* dedicated to select papers of SODA 2002.

GP91. Karni Gilon and David Peleg. Compact Deterministic Distributed Dictionaries. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 81–94, 1991.

HJS+03. Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 113–126, Seattle, WA, USA, March 2003.

JC94. Theodore Johnson and Adrian Colbrook. A Distributed, Replicated, Data-Balanced Search Structure. In *International Journal of High Speed Computing*, volume 6, pages 475–500, 1994.

KK03. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

KM05. Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144, New York, NY, USA, 2005. ACM Press.

KR04. David R. Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *ACM SPAA*, 2004.

MP84. J. Ian Munro and Patricio V. Poblete. Fault Tolerance And Storage Reduction In Binary Search Trees. *Information and Control*, 62(2/3):210–218, August 1984.

MRS01. Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results, 2001.

Pug90. William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

RRHS04. Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief Announcement: Prefix Hash Tree. In *Proceedings of ACM PODC*, St. Johns, Canada, July 2004.

SFG+02. Jared Saia, Amos Fiat, Steven Gribble, Anna Karlin, and Stefan Saroiu. Dynamically Fault-Tolerant Content Addressable Networks. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *Lecture Notes in Computer Science*, pages 270–279, Cambridge, MA, USA, March 2002.

SG88. Dennis Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, pages 53–90, 1988.

SMLN+03. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11:17–32, February 2003.