home

Building A Skiplist

06 Dec 2013

In previous posts, we've spent a lot of time talking about and using various fundamental data structures, such as arrays, linked lists and hashtables. While simple, these data structures tend to be specialized. For example, arrays can efficiently access values by index, but aren't efficient when searched or when changing/growing. It must be a law that data structures capable of a few O(1) operations must suffer O(N) (or worse) for all other operations.

What options are available when you need to perform a variety of operations against your data? Specifically, what if you want to efficiently lookup and iterate values without sacrificing the ability to insert and remove items? The most common example of this is a database index. The most common solution is some variation of a B-Tree. B-Trees are efficient because they maintains the order of values. Given sorted values, we can find, remove or insert values in Log(N) time thanks to divide and conquer. Find a value out of 1 000 000? 20 operations worst case. Find a value out of 100 000 000 items? 26 operations worst case. It's much closer to O(1) than O(N).

Although not a Tree, skiplists are a popular alternative which share the same performance characteristics while being easier to implement. Like the difference between various Tree implementations, skiplists have their own unique behaviors, but we'll leave that for another time. For now, we'll look at understanding how skiplist work and building one.

Skiplists are built using linked lists. However, while linked lists are only concerned with next and prev, skiplists are also concerned with up and down. In fact, thinking of a skiplist in terms of a multiple linked lists grouped vertically is a good way to start. Let's visualize that:

```
HEAD                                                        TAIL
 |----------------------------->e-----------------------------|
 |                              |                             |
 |----------------->c---------->e---------->g-----------------|
 |                  |           |           |                 |
 |<---->a<--->b<--->c<--->d<--->e<--->f<--->g<--->h<--->i<--->|
```

Looking at the above, you should see the general approach you'd take to find an item within the skiplist. Start at the top layer and divide and conquer. Looking for "g", we'd:

1. Start at level 3 (the top), is next (e) smaller or equal than g? Yes, move forward
2. Still on level 3, is next (nil) smaller or equal than g? No, move down
3. Now on level 2 (e), is next (g) smaller or equal than g? Yes. Found!

You start at the highest level moving forward as long as the next value is less than your target. When the next value is greater, you move down and repeat the process. You'll eventually either get your item or reach the tail. Either way, given a perfectly distributed skiplist, every operation halves the possible haystack, resulting in a Log(N) search.

The lowest level of a skiplist contains all elements. It's also the only level which needs to be implemented as a doubly linked list. Why? When searching we always move forward and down. There's never a need to move backwards because we'll never overshoot (if we see that next is larger, we move down). Having a doubly linked list at the lower level means that we can iterate through all items, forward or backwards, in O(N).

You might be thinking that inserts are going to be tricky since we have to take care of putting values at the correct level. It's true that improperly distributed, skipslists will perform no better than a linked list. However, ensuring proper distribution isn't hard, we just roll the dice.

100% of the values will be placed on level 0. How many should also be promoted to level 1? Ideally, 50%. How many should be promoted to level 2? 25%. Level 3? 12.5%. Level 4? 6.25% and so on. Given anything but an insignificant number of inputs, you will end up with a well distributed structure (this can be proved).

Let's look at an insert function to really wrap our minds around it. First, we need to decide the maximum number of levels our skiplist will support. With a height of 32, we can expect a good distribution with up to 2^32 values. Next, like all linked lists, we need to create a `head`. However, unlike a normal `head`, this one is vertical-aware:

```go
const maxHeight = 32

type Skiplist struct {
  height int
  head *SkiplistNode
}

type SkiplistNode struct {
  value string
  next []*SkiplistNode
  prev *SkiplistNode
}

func New() *Skiplist {
  head := &SkiplistNode {
    value: "",
    next: make([]*SkiplistNode, maxHeight),
  }

  return &Skiplist {
    height: 0,
    head: head,
  }
}
```

What exactly is this? First we have the `Skiplist` structure which, like a linked lists, has a `head` but also has the current height. Next we have the `SkiplistNode` which, like a linked list node, has a `value` and a reference to the `prev` node. Our `SkiplistNode` also has a `next` reference, but it supports multiple values, one per level. 50% of all our nodes will only have a single `next` reference, 25% will have 2. 12.5% will have 3 and so on. Our head is our entry point into the entire structure, so it needs to be able to support 32 `next` references.

So far all we have is an empty structure. Let's add a value. The first thing we need to do is figure out what level it'll go up to. In theory, we could pick any level from 0-31. However, unless we know for sure that we'll be dealing with 4 billion values, it's best to limit ourselves to 0 - (height+1). In other words, we limit our growth to 1 level at a time. A common way to figure the level is to roll the dice in a loop:

```
func (s *Skiplist) Set(value string) {
  level := 0
  // Int31n is exclusive, so we'll either get 0 or 1
  for ; rand.Int31n(2) == 1 && level < maxHeight; level++ {
    if level > s.height {
      s.height = level
      break
    }
  }
  ...
}
```

What are the chances of rolling a single 1? 50%. Two in a row? 25%. Three? 12.5%. Since our skiplist's height is initially 0, and since we break once our level is higher than the current height, the first value can only be promoted to level 1 (and it only has a 50% chance of such a promotion). There are slightly more efficient ways to calculate the level, but none are nearly as illustrative.

At this point, inserting the value is a matter of walking through our structure (next and down) and inserting the value at the right position (both in terms of the vertical position and the horizontal one):

```
  // the node we'll be inserting
  node := &SkiplistNode {
    value: value,
    // +1 because if we're at level 0, we need 1 slot
    // for our next. Level 1? we need 2 slots, and so on
    next: make([]*SkiplistNode, level+1),
  }

  //start at the top of the head
  current := s.head
  for i := s.height; i >= 0; i-- {
    //move next until we either hit the end OR...
    for ; current.next[i] != nil; current = current.next[i] {
      next := current.next[i]
      // ...OR the next node is larger than our new value
      if next.value > value { break }
    }

    if i > level { continue }
    node.next[i] = current.next[i]
    current.next[i] = node
    node.prev = current
  }
```

The important thing to realize here is that, although we search from the very top level of our skiplist, we don't necessarily insert at that level. It's entirely possible that our skiplist has 15 levels, but this particular node only goes on level 0. But it's important that we search from the very top so that we can quickly find the right position in log(N) time. This is why we have the `if i > level { continue }` code. Once we've hit that line, we know that this is the right location at this level (either because the next node is nil or because we've the next node is larger), but we might not actually need to insert at this level. The effort of reaching this spot wasn't wasted though, because its from this point that our next iteration goes down a level and continues its search.

The last three lines are just about hooking up the linked list at a particular level. Nothing fancy there. But looking at skiplist operations as two distinct steps is helpful:

1. Find the position per-level
2. Do a normal linked list operation at that level (if appropriate)

In fact, there's no point in looking at a get or delete operation, because they follow the exact same pattern. Once you know how to walk through the structure to find a node (either the actual node you want, in the case of delete, get or update, or an adjacent node, as above, for insert) it's just a normal linked list operation.

What might be interesting is displacing a skiplist. For this, I'll send you over to the above implementation on Go's playground. Note that the output is fixed, but you can change the `rand.Seed(8)` near the bottom to see different outputs.

A few final words. First, for a small set, you'll probably get seemingly poor distribution. In such cases, linearly scanning an array is probably best anyways. Second, not only can you adjust the maximum height of the skiplist, but you can also adjust the rate of growth. Instead of `rand.Int31n(2) == 1` we could use `rand.Int31n(3) == 1` to make it grow much less, or, alternatively, change it so that it grows much faster. That said, 32 levels and 50% chance to level up, seems standard.

Hope that was fun.

Comments for this thread are now closed. ×

3 Comments Karl Seguin's Blog 1 Login

Recommend Share Sort by Oldest

Malcolm Greaves • 4 years ago

Nice post!

1 • Share ›

Dan Breczinski • 4 years ago

I'm curious if the efficiency of operations changes if you change the rate of growth. What would happen to performance if the chance of promoting a node on insert was something like 10%, 1%, 0.1%? It seems, given a large enough data set things would average out. You would spend less time getting to the bottom level but more time traversing the bottom level.

The main motivation for doing this would be to save memory. If you didn't need to traverse backwards, you could implement the bottom level as a singly linked list then, you would approach needing only one pointer per node as you decreased the probability of node promotion. If only 1/100 nodes was promoted only 1/100 nodes would require more than one pointer. Assuming performance would still be Log(N), decreasing the probability of promotion would have similar performance as a B-Tree but less memory usage since each node in a B-Tree has two pointers.

• Share ›

Karl Seguin Mod → Dan Breczinski • 4 years ago

Check out http://www.cl.cam.ac.uk/tea... and search for "Choosing P" :)

• Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy