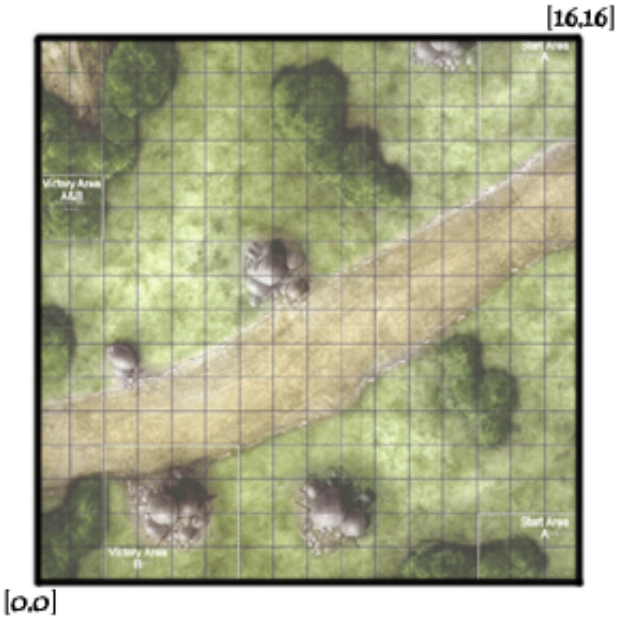
Snail in a Turtleneck

« NoSQL vs. the world
PS1++ »

Mongo in Flatland

MongoDB's geospatial indexing lets you use a collection as a map. It works differently than "normal" indexing, but there's actually a nice, visual way to see what geospatial indexing does.

Let's say we have a 16×16 map; something that looks like this:

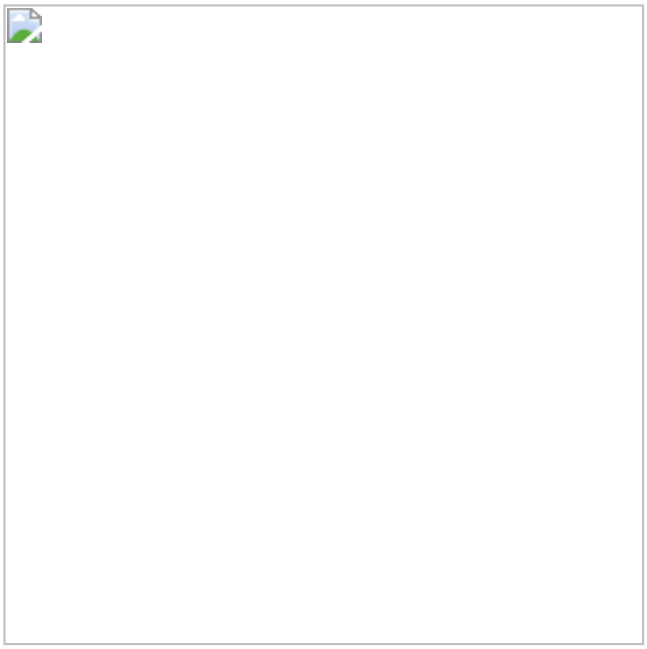


All of the coordinates in our map (as described above) are somewhere between [0,0] and [16,16], so I'm going to make the *min* value 0 and the *max* value 16.

```
db.map.ensureIndex({point : "2d"}, {min : 0, max : 16, bits : 4})
```

This essentially turns our collection into a map. (Don't worry about *bits*, for now, I'll explain that below.)

Let's say we have something at the point [4,6]. MongoDB generates a *geohash* of this point, which describes the point in a way that makes it easier to find things near it (and still be able to distribute the map across multiple servers). The geohash for this point is a string of bits describing the position of [4,6]. We can find the geohash of this point by dividing our map up into quadrants and determining which quadrant it is in. So, first we divide the map into 4 parts:



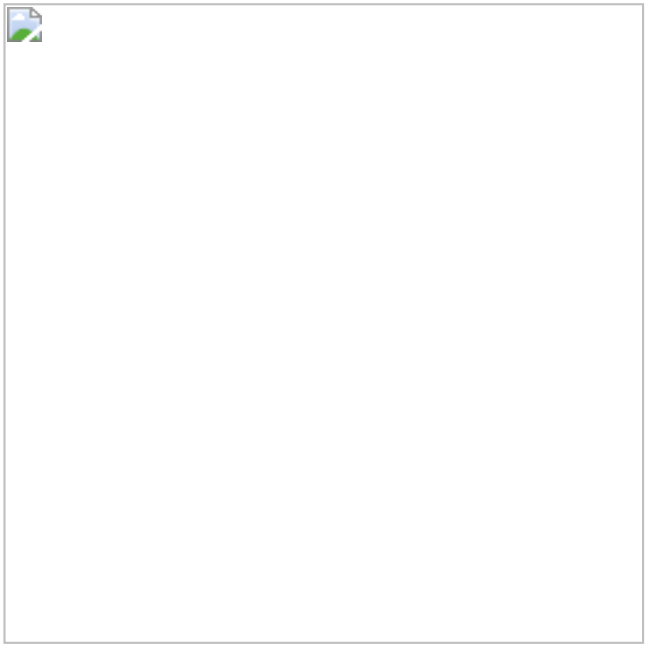
This is the trickiest part: each quadrant can be described by two bits, as shown in the table below:

01	11
00	10

[4,6] is in the lower-left quadrant, which matches 00 in the table above. Thus, its geohash starts with 00.

Geohash so far: 00

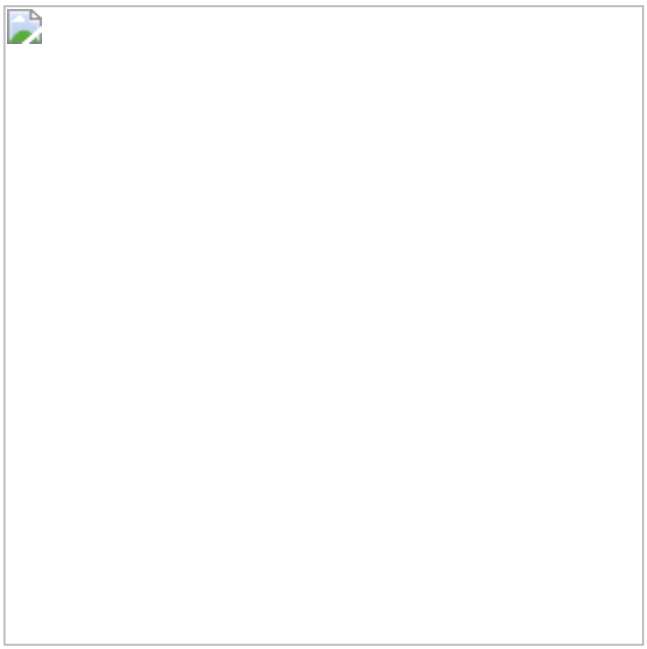
Now we divide that quadrant again:



[4,6] is now in the upper-right quadrant, so the next two bits in the geohash are 11. Note that the bottom and left edges are included in the quadrant, the top and right edges are excluded.

Geohash so far: 0011

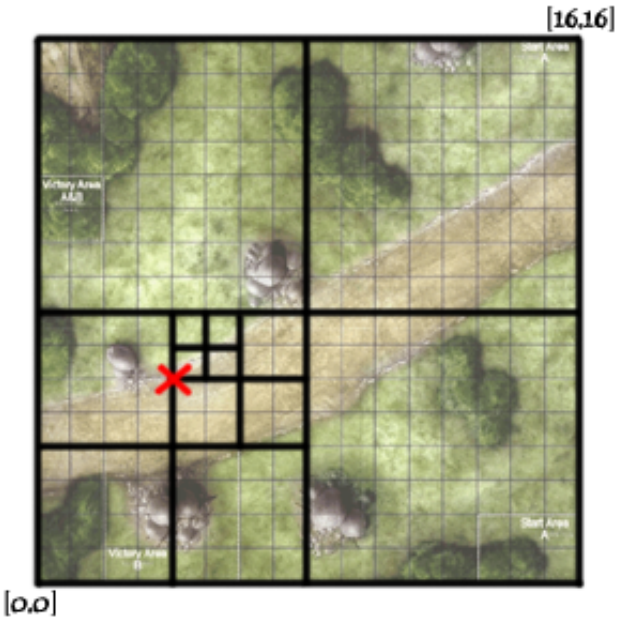
Now we divide that quadrant again:



[4,6] is now in the upper-left quadrant, so the next two bits in the geohash are 01.

Geohash so far: 001101

Now we divide that quadrant again:



[4,6] is now in the lower-left quadrant, so the next two bits in the geohash are 00.

Geohash so far: 00110100

You may wonder: how far do we keep dividing? That's exactly what the *bits* setting is for. We set it to 4 when we created the index, so we divide into quadrants 4 times. If we wanted higher precision, we could set *bits* to something higher.

You can check your math above by using the *geoNear* command, which returns the geohash for the point you're search near:

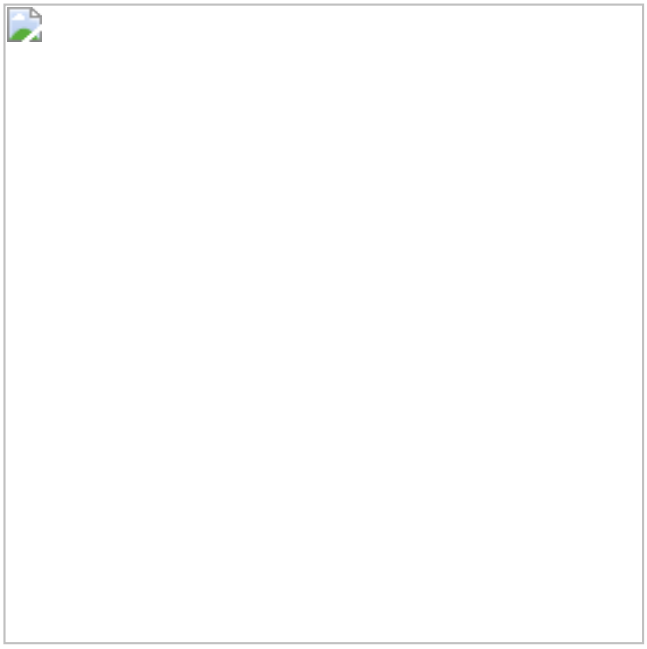
```
> db.runCommand({geoNear : "map", near : [4,6]})
{
        "ns" : "test.map",
        "near" : "00110100",
        "results" : [ ],
        "stats" : {
                "time" : 0,
                "btreelocs" : 0,
                "nscanned" : 0,
                "objectsLoaded" : 0,
                "avgDistance" : NaN,
                "maxDistance" : -1
        },
        "ok" : 1
}
```

As you can see, the "near" field contains exactly the geohash we'd expect from our calculations.

The interesting thing about geohashing is that this makes it easy to figure out what's near us, because things are sorted according to their position on the map: every document with a *point* geohash starting with 00 is in the lower-left quadrant, every point starting with 00111111 is very near the middle, but in the lower-left quadrant. Thus, you can eyeball where a point is by looking at its geohash.

Bits and Precision

Let's say a wizard casts a ring of fire around him with a radius of 2. Is the point [4,6] caught in that ring of fire?



It's pretty obvious from the picture that it isn't, but if we look at the geohash, we actually can't tell: [4,6] hashes to 00110100, but so does [4.9, 6.9], and any other value in the square between [4,6] and [5,7]. So, in order to figure out whether the point is within the circle, MongoDB must go to the document and look at the actual value in the *point* field. Thus, setting *bits* to 4 is a bit low for the data we're using/queries we're doing.

Generally you shouldn't bother setting *bits*, I've only set it above for purposes of demonstration. *bits* defaults to 26, which gives you approximately 1 foot resolution using latitude and longitude. The higher the number of bits, the slower geohashing gets (conversely, lower bit values mean faster geohashing, but more accessing documents on lookup). If you're doing particularly high or low resolution queries, you might want to play around with different values of *bits* (in dev, on a representative data set) and see if you get better performance.

Thanks to Greg Studer, who gave a geospatial tech talk last Friday and inspired this post. (Every Friday, a 10gen engineer does a tech talk on something they're working on, which is a really nice way to keep up with all of the cool stuff coworkers are doing. If you're ever running an engineering department, I highly recommend them!)

This entry was posted Wednesday, June 8th, 2011 at 7:54 am and is filed under MongoDB. You can skip to the end and leave a response. Pinging is currently not allowed.

Follow me on Twitter

- # Email Subscription

Email Subscription

Your email here

Subscribe

- # order my books

MongoDB
Kristina Chodorow
Best Price $24.30
or Buy New $36.92
Buy from amazon.com

Privacy Information

Scaling MongoDB
Kristina Chodorow
Best Price $6.78
or Buy New $28.49
Buy from amazon.com

Privacy Information

- **about me**

I'm a software engineer at Google in NYC. I have done a lot of work on MongoDB and enjoy reading and cartooning more...

- **Search**

- **Related Posts**

No related posts.
