When you’re working with a large dataset, one issue is optimizing search and selection of elements in a region. Let’s say you’re working with a set of data with xy coordinates (anything that’s laid out on a plane or screen). You’ve seen enough examples in this book to know that this may be a scatterplot, points on a map, or any of a number of different things. When you have data like this, you often want to know what datapoints fall in a particular selected region.

This is referred to as spatial search (and notice that “spatial” in this case doesn’t refer to geographic, but rather space in a more generic sense). The quadtree functionality is a spatial version of d3.nest, which you used in chapter 5 and chapter 8 and will use again in chapter 12 to create hierarchical data. Following the theme of this chapter, you’ll get started by creating a big dataset of random points and render them in SVG.

Generating random xy data

For your third random data generator, you don’t have to put nearly the kind of work in that you did for the first two. In the following listing, all you do is create 3000 points with random x and y coordinates.

Listing 1 Xy data generator

```javascript
var sampleData = d3.range(3000).map(function(d) {
    var datapoint = {};
    datapoint.id = "Sample Node " + d;
    datapoint.x = Math.random() * 500;
    datapoint.y = Math.random() * 500; #a
    return datapoint;
});

d3.select("svg").selectAll("circle")
    .data(sampleData)
    .enter()
    .append("circle")
    .attr("class", "xy")
    .attr("r", 3)
```

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/meeks/
.attr("cx", function(d) {return d.x})
.attr("cy", function(d) {return d.y});

**Because you know the fixed size of your canvas, you can hardwire this**

As you may expect, the result of this code, shown in figure 11.11 is a bunch of pink circles scattered randomly all over your canvas.

![Figure 1 3000 randomly placed points represented by pink SVG <circle> elements](image)

**Xy brushing**

Now you’ll create a brush to select some of these points. Allow brushing along both x- and y-axes. Then you can drag a rectangle over any part of the canvas. In listing 2, you can see how quick and easy it is to add a brush to your canvas. You’ll also add a function to highlight any circles in the brushed region. In this example you use d3.scale.identity for your .x() and .y() selectors. All d3.scale.identity does is create a scale where the domain and range are exactly the same. It’s useful for times like these when the function operates with a scale but your scale domain directly matches the range of your graphical area.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/meeks/](http://www.manning.com/meeks/)
Listing 2 Xy brushing

```javascript
var brush = d3.svg.brush()
  .x(d3.scale.identity().domain([0, 500])) #a
  .y(d3.scale.identity().domain([0, 500]))
  .on("brush", brushed);

d3.select("svg").call(brush)

function brushed() {
  var e = brush.extent();
  d3.selectAll("circle")
    .style("fill", function (d) {
      if (d.x >= e[0][0] && d.x <= e[1][0] &&
        d.y >= e[0][1] && d.y <= e[1][1]) #b
        { return "darkred"; #c
          }
        else { return "pink"; #d
          }
    });
};

#A Because you aren’t going to adjust scale settings, you can define them inline
#B Tests to see if the data is in your selected area
#C Colors the points in the selected area dark red
#D Colors the points outside the selected area pink

With this brushing code, you can now see the circles in the brushed region, as shown in figure 2.
```
This works, but it’s terribly inefficient. It checks every point on the canvas without using any mechanism to ignore points that might be well outside the selection area. Finding points within a prescribed area is an old problem that has been well explored. One of the tools available to answer that question quickly and easily is a quadtree. You may ask, what is a quadtree and what do I use it for?

A quadtree is a method for optimizing spatial search by dividing a plane into a series of quadrants. You then divide each of those quadrants into quadrants, until every point on that plane falls in its own quadrant. By dividing the xy plane like this, you nest the points you’ll be searching in such a way that you can easily ignore entire quadrants of data without testing the entire dataset.

Another way to explain a quadtree is to show it. That’s what this information visualization stuff is for, right? Figure 3 shows the quadrants that a quadtree produces based on a set of point data.
Creating a quadtree with xy data of the kind you have in your dataset is easy, as you can see in the following listing. You set the x and y accessors like you do with layouts and other D3 functions.

**Listing 3 Creating a quadtree from xy data**

```javascript
var quadtree = d3.geom.quadtree()
 .extent([[0,0], [500,500]]) #a
 .x(function(d) {return d.x})
```

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/meeks/](http://www.manning.com/meeks/)
A quadtree is a tree data structure with a series of nested boxes that subdivide the space it represents. Each node in the tree represents a region of space. A quadtree is defined as an array of upper left and lower right points:

```javascript
var quadIndex = quadtree(sampleData); //c
```

Accessors pointed at your data's xy format

After creating a quadtree, you create the index by passing your dataset to it:

```javascript
quadIndex.visit(function(node, x1, y1, x2, y2) { //b
    if (node.point) { //c
        if (node.point.x >= e[0][0] && node.point.x <= e[1][0]
            node.point.y >= e[0][1] && node.point.y <= e[1][1]) { //d
            node.point.selected = true;
        }
    }
    return x1 > e[1][0] || y1 > e[1][1] || x2 < e[0][0] || y2 < e[0][1]; //e
});
```

You can then use that dataset's .visit() function for quadtree-optimized searching:

```javascript
d3.selectAll("circle")
    .filter(function(d) {
        return d.selected;
    })
    .style("fill", "darkred"); //f
};
```

The .visit() functionality replaces your test in a new brush function, as shown in listing 4. First, I'll show you how to make it work in listing 4. Then I'll show you that it works in figure 4, and I'll explain how it works in detail. This isn’t the usual order of things, I realize, but with the quadtree, it makes more sense if you see the code before explaining its exact functionality.

**Listing 4 Quadtree-optimized xy brush selection**

```javascript
function brushed() {
    var e = brush.extent();

    d3.selectAll("circle")
        .style("fill", "pink")
        .each(function(d) {d.selected = false}); //a
    quadIndex.visit(function(node, x1, y1, x2, y2) { //b
        if (node.point) { //c
            if (node.point.x >= e[0][0] && node.point.x <= e[1][0]
                node.point.y >= e[0][1] && node.point.y <= e[1][1]) { //d
                node.point.selected = true;
            }
        }
        return x1 > e[1][0] || y1 > e[1][1] || x2 < e[0][0] || y2 < e[0][1]; //e
    });
    d3.selectAll("circle")
        .filter(function(d) {
            return d.selected;
        })
        .style("fill", "darkred"); //f
};
```

# The .visit() functionality replaces your test in a new brush function
# First, I’ll show you how to make it work in listing 4
# Then I’ll show you that it works in figure 4
# And I’ll explain how it works in detail
# This isn’t the usual order of things, I realize, but with the quadtree, it makes more sense if you see the code before explaining its exact functionality

- **#a** Sets all circles to pink, and gives each a selected attribute to designate which are in your selection
- **#b** Calls .visit()
- **#c** Checks each node to see if it’s a point or a container
- **#d** Checks each point to see if it’s inside your brush extent and sets selected to true if it is
- **#e** Checks to see if this area of the quadtree falls outside your selection
- **#f** Shows which points were selected

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/meeks/](http://www.manning.com/meeks/)
The results are impressive and much faster. In figure 11.14, I increased the number of points to 10,000 and still got good performance. (But if you’re dealing with datasets that large, I recommend switching to canvas, because forcing the browser to manage all those SVG elements is going to slow things down.)

Figure 4 Quadtree-optimized selection used with a dataset of 10,000 points

How does it work? When you run the visit function, you get access to each node in the quadtree, from the most generalized to the more specific. With each node, which you access in listing 11.16 as node, you also get the bounds of that node (x1, y1, x2, y2). Because nodes in a quadtree can either be the bounding areas or the actual points that generated the quadtree, you have to test if the node is a point.
and, if it is, you can then test if it’s in your brush bounds like you did in your earlier example. The final piece of the visit function is where it gets its power, but it’s also the most difficult to follow:

```
return x1 > e[1][0] || y1 > e[1][1] || x2 < e[0][0] || y2 < e[0][1]
```

Left of node greater than right of selection  Bottom of node greater than top of selection  Right of node less than left of selection  Top of node less than bottom of selection

The visit function looks at every node in a quadtree, unless you return true, in which case it stops searching that particular quadrant and all its child nodes. So you test to see if the node you’re looking at (represented as the bounds x1,y1,x2,y2) is entirely outside the bounds of your selection area (represented as the bounds e[0][0], e[0][1], e[1][0], e[1][1]). You create this test to see if the top of the selection is below the bottom of the node’s bounds, if the bottom of the selection is above the top of the node’s bounds, if the left side of the selection is to the right of the right side of the node’s bounds, or if the right side of the selection is to the left of the left side of the node’s bounds. That may seem a bit hard to follow (and sure takes up more time as a sentence than it does as a piece of code), but that’s how it works.

You can use that visit function to do more than optimized search. I’ve used it to cluster nearby points on a map (http://bl.ocks.org/emeeks/066e20c1ce5008f884eb) and also to draw the bounds of the quadtree in figure 3.