Different approaches to increase performance of the d3.js applications

The d3 applications are dependent on data which can be large or small. When we use small amount of data, the applications can smoothly run. But when we deal with large amount of data, the performance issue comes into picture. When the data is large the application rendering and interaction becomes slow. In such cases, it is better to pre-render the application on the server.

There are various approaches to deal with performance issues. This paper explains two different approaches which increase the performance if large data is involved. These two approaches are:

• Node.js Approach: Separate the JavaScript files and load them using node.js

• Separation Approach: Separate the Html, CSS, JavaScript files and load them using simple JavaScript selector

Node.js Approach

The node.js approach is an efficient but complex approach to increase the application performance. In this approach we need to download the latest version of node.js and its dependencies. We need to install different dependencies such as d3, jsdom and fs.

When we are dealing with huge data, the best way is to render the data and the visuals on the server side and send the output to the user to interact. Therefore, there is less time taken by the application to load the data and the visuals on the client side.

In the node.js approach we need to separate all the JavaScript files and store them as separate .js files. After that, we can load these files using the special JavaScript files that have all the Html code and references to the JavaScript files. We can run this special .js file from the node.js command prompt. We will see each step in detail.

There are four different steps involved in this approach:

1. Generating HTML file using jsdom

2. Render the data visualization with d3 node.js module

3. Generate the d3.js client side script

4. Using the result

1. Generating HTML file using jsdom:

To follow this step, we need to install all the dependencies such as d3, jsdom and fs. To install these dependencies use the command

$ npm install d3

$ npm install jsdom

$ npm install fs

The special .js file mentioned earlier will look like the below code:

var d3 = require('d3')

, jsdom = require('jsdom')

, htmlStub = '<html><head></head><body><div id="dataviz-container"></div><script src="js/d3.v3.min.js"></script></body></html>'

// html file skull with a container div for the d3 dataviz

// pass the html stub to jsDom

jsdom.env({ features : { QuerySelector : true }, html : htmlStub

, done : function(errors, window) {

//write d3.js code here

}

})

In the above code there are functions to generate the HTML code using jsdom. So before we start, we need to import all the dependencies. It can be accomplished using the require() method.

In the above code we write the HTML elements to be generated which are stored in the htmlstub variable. This variable is passed to the jsdom.env() method.

The code above creates an html document on the server, kept in memory. We have included html tags, a container div for the svg, as well as a script loading d3.js on the client.

Using jsdom we can manipulate the html file using d3.js just as if we were doing it on the client side. To do so, we have to use jsdom’s querySelector function and pass its result to d3.js.

We can now select the div elements using simple javascript as follows:

var el = window.document.querySelector('#dataviz-container')

, body = window.document.querySelector('body')

2. Render the data visualization with the d3 node.js module:

In this step we can write the code to generate graph. Here we will use d3.js to generate different visualization on the page.

d3.select(el)

.append('svg:svg')

.attr('width', 600).attr('height', 300)

.append('circle')

.attr('cx', 300).attr('cy', 150).attr('r', 30).attr('fill', '#26963c')

Remember, the d3 code above is executed on the server. The html document we have generated contains the svg circle, and not the lines above.

3. Generate the d3.js client side script:

In this step, we will write the code for client interaction. We need to inject the client script inside the html. In our example, the client-side script will select the circle by its Id and transition its fill color to orange.

var circleId = 'a2324' // say, this value was dynamically retrieved from a database

// append the svg to the selector

d3.select(el)

.append('svg:svg')

.attr('width', 600).attr('height', 300)

.append('circle')

.attr('cx', 300).attr('cy', 150).attr('r', 30).attr('fill', '#26963c')

.attr('id', circleId) // we assign the circle to an Id here

// write the client-side script manipulating the circle

var clientScript = "d3.select('#" + circleId + "').transition().delay(1000).attr('fill', '#f9af26')"

// append the script to page's body

d3.select(body)

.append('script')

.html(clientScript)

4. Using the result

In the final step we need to generate the actual processed html file. It can be accomplished using fs module. We can write the final processed html code with the svg in html file and save it as index.html. the user can then open this file to see and interact with the visualization. The code will look like :

// save result in an html file

var fs = require('fs')

, svgsrc = window.document.innerHTML

fs.writeFile('index.html', svgsrc, function(err) {

if(err) {

console.log('error saving document', err)

} else {

console.log('The file was saved, open index.html to see the result')

}

});

In node.js approach the html file is generated dynamically. Also all the loading of data, rendering and processing done on the server so the application becomes fast.

This approach our application is dependent on lot of libraries which we need to install. Sometimes it gives an error and our application can’t run. Also, this method is very complex.

Separation Approach

This is the second approach to increase application performance. This method is very simple. We need to separate each file i.e. HTML, CSS and JavaScript files. Give the reference of all the files in the HTML file and run the application.

There are very few steps in this approach:

• Separate each JavaScript library in separate folder.

• Keep all the css files in separate directory and name it css.

• Keep the data in separate directory and name it data.

• Make separate JavaScript file for different graph e.g. if you are using Line chart, Bar chart, Scatter plot then you must write separate JavaScript file for all of them.

• Create and index.html file and give the reference of all the JavaScript libraries, css and graph generating JavaScript files into it as follows:

<link rel="stylesheet" type="text/css" href="components/bootstrap/dist/css/bootstrap.css"/>

<link rel="stylesheet" type="text/css" href="components/bootstrap/dist/css/bootstrap-theme.css"/>

<link rel="stylesheet" type="text/css" href="css/style.css"/>

<script src="components/jquery/dist/jquery.js"></script>

<script src="components/bootstrap/dist/js/bootstrap.js"></script>

<script src="components/lodash/dist/lodash.js"></script>

<script src="components/moment/moment.js"></script>

<script src="components/d3/d3.js"></script>

<script src="http://d3js.org/d3.hive.v0.min.js"></script>

<script src="js/forcedirected.js"></script>

<script src="js/hive.js"></script>

<script src="js/dendrogram.js"></script>

<script src="js/matrix.js"></script>

<script src="js/crossfilter.js"></script>

<script src="js/main.js"></script>

• Now add the html code i.e. container to hold the graph into it and run the application.

This is very simple approach to increase the application performance. Also it is not dependent on libraries such as jsdom and fs. So we do not need to install them.