Heat Map

Problem Statement :

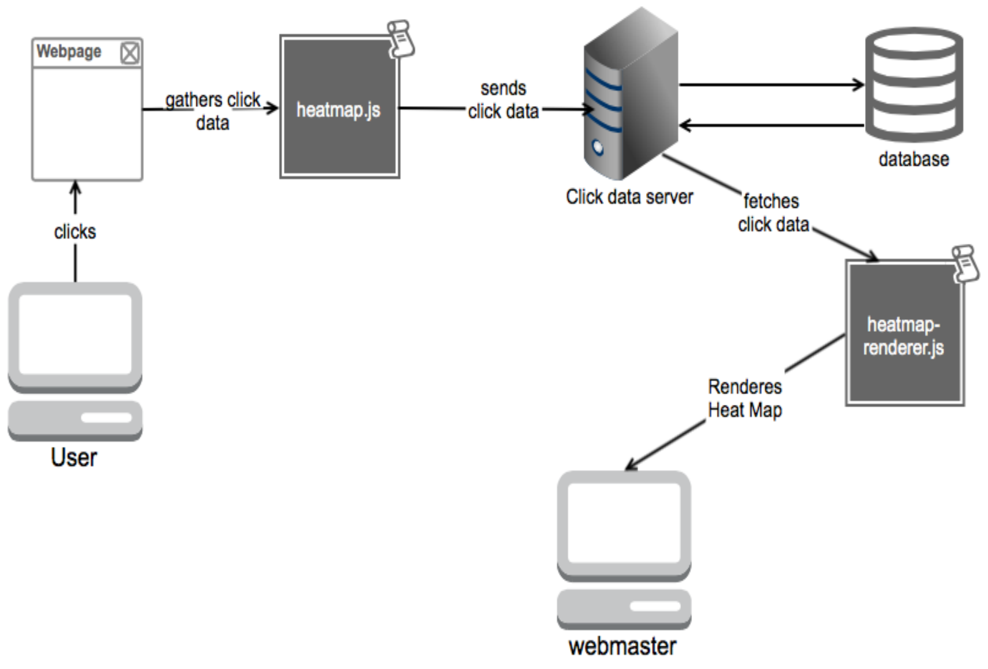
Design a system that can gather click data from a webpage and show it later in form of a Heat map

Solution :

Since click data on web page can be detected on the client side. So this solution has to be programmed in client side script (obviously Javascript). This solution involves server dedicated to store and serve click data to client side scripts for running this solution. This Heat-map system is composed of 2 client side scripts and 1 server.

- **heatmap.js** : This script is included in webpage from which click data needs to be fetched. Its use is to collect meaningful click data from the client side and send it to the server.
- **heatmap-renderer.js** : This script fetches click data from the server and renders heat map of the webpage for the owner/admin.
- **Click Data Server** : Purpose of this server is to collect click data from the heatmap.js script , store it and server the data to heatmap-renderer.js on demand , in form of chunks , making it a scalable solution.

This is the holistic view of the system.



This folder contains both of these js files along with the working example. For the sake of demonstration, both the collection and rendering of heat map is done on same web page.

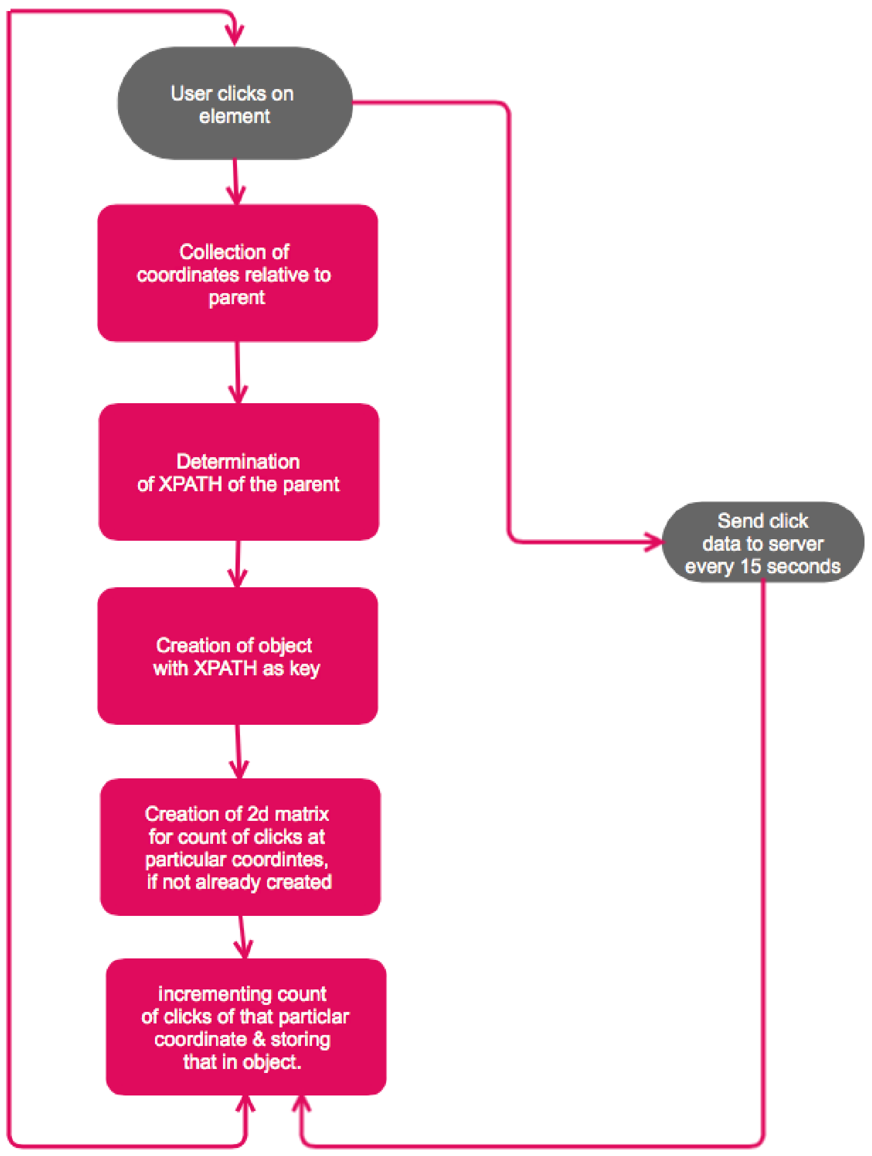
Working :

The engine of this heat map systems lies in heatmap.js which collects the click data from the webpage. As a very basic step heatmap.js creates a heatmap class and instantiates an object which does all the work. The first action this object takes is , it attaches a click event listener on whole body element. Remaining part is done by the event listener.

Whenever click occurs on the page, event listener stores the reference of the DOM element on which click is done. Since there is no guarantee that that DOM element will be having a class or id , by which it can be identified.
So the only way the DOM element can be identified is by XPATH. heatmap.js generates the XPATH of the DOM element in a smart manner. (In the nutshell , it goes to the nearest parent that can be identified , by class or id or name, and traces the route back to the element). After identifying the element by XPATH, it creates an object with XPATH as its key and store the count of the clicks in that element as well as 2D matrix of count of clicks at particular coordinates. Value of *clicks[x][y]* gives value of count of clicks at (x,y) in that element. Interesting thing here is that , it stores coordinates relative to the parent element, so that coordinates can be reproduced in responsive website.
Parallely heatmap.js periodically (every 15 seconds , customizable) sends click data to the server in form of ajax calls. In this manner , we ensure that the data if lost , will only be of 15 seconds, hence we don't lose much data. Also since query is run every 15 seconds, the data sent to the server won't be big in size and hence it won't cause bandwidth issues. (in the example file provided , heatmap.js sends whole data , but it can be tweaked easily to allow data transfer of only 15 seconds of activity).

The server in this case has to be designed in such a way , so that it concatenates the data into existing data , and not save it in database every 15 seconds. It should keep the data in working memory and only save it in database at least after 30 mins. In this way server won't overwhelm the database.

Following diagram explains the process quickly


User clicks on element
Collection of coordinates relative to parent
Determination of XPATH of the parent
Creation of object with XPATH as key
Creation of 2d matrix for count of clicks at particular coordintes, if not already created
incrementing count of clicks of that particlar coordinate & storing that in object.
Send click data to server every 15 seconds

Heatmap-renderer creates 2 overlays, which are hidden by default and are toggled by fixed position buttons on the page. Height and Width of overlays are dynamically set to the dimensions of the page. Renderer creates spots on the overlay at particular position where click occurred in the DOM element. This done with the help of relative coordinates. These spots on the overlay are basically DIVS styled accordingly to give appearance of spots.
Colors of the spots gives intensity of the clicks on the particular region. In this case **RED** region is the area of maximum clicks, **GREEN** which is in between maximum and minimum and **Blue** where minimum number of clicks occurred.

RGB calculation is done dynamically according to max and min # of clicks. First the half value id calculated as follows :

```
var halfmax = (min+max)/2;

var b = parseInt(max(0, 255*(1 - parseFloat(value)/halfmax)));
var r = parseInt(max(0, 255*(parseFloat(value)/halfmax - 1)));
var g = 255 - b - r;
```

This color calculation could be done in alternative method in which the colors only comprised of red and blues , where most red one is the region of highest clicks and most blue with min number of cliks. For this alternative method , we have to scale the range of clicks to [0,1]. and then assign RGB as :

```
var r = 255 * (( value - min)/(max - min));
var b = 255 * (1 - (( value - min)/(max - min)));
var g = 0 ;
```

Clickmap functionality (included in heatmap) , outlines the DOM element in the overlay and displays the total number of clicks in it.

Issues targeted :

- **System Integration** : Since the click data collection script is just a single javascript file. This system can be integrated by just including the script in webpage. Its dependencies just include Jquery (which is pretty standard). Since the rendering and data collection scripts are separate, both work independently without intervening each others functionality.

- **Functioning in responsive websites** : heatmap.js records click data in respect to the parent element. This means that even if element changes position in responsive website, click data will be associated with that element only and rendering of the heat map won't be affected.

- **Scalability** : For scalability the heatmap-rederer.js doesn't fetch the whole click data at once and then process the heatmap. Instead heatmap data is returned in chunks from the server and the chunks received are immediately processed in heatmap. Here chunks from the server should contain complete data for DOM element , so that each chunk contains whole data of DOM element so that it can be processed immediately irrespective of other DOM elements. While the script is still receiving/waiting for other chunks 'waiting sign ' or loading bar is displayed , so that user knows that data is still being fetched. Thus in nutshell heatmap-renderer.js first queries server for number of chunks in which data would be returned along with min and max value of clicks (for color calculation) , and then make that many number of ajax calls to fetch chunks from the server , and immediately process the chunks received to generate their heatmap. This way huge data of even 1 million clicks would be handled in efficient way , and user would able to see the heatmap of already fetched data.