DBMS DA214

Assignment 3
-By Rishita Agarwal

-220150016

---

https://observablehq.com/@d3/walmarts-growth?intent=fork

https://www.kaggle.com/datasets/ssarkar445/us-chronic-disease-indicators

Question 1

The existing set has a dataset that helps in identifying the growth of new Walmart outlets in different parts of the USA. Repeat the task on your local machine and note down the locations of the functions that do the task.

https://observablehq.com/@d3/walmarts-growth?intent=fork

This is the link for the observable.

Explanation of the code:

data = (await FileAttachment("walmart.tsv").tsv()) .map(d => { const p = projection(d); p.date = parseDate(d.date); return p; }) .sort((a, b) => a.date - b.date)

This line is used to load the data from the walmart.tsv present in the environment.Convert geographic coordinates to pixel coordinates on the map using a `projection` function.

parseDate = d3.utcParse("%m/%d/%Y")

Defines `parseDate` as a function that uses D3's `utcParse` method to convert date strings in the format of "month/day/year" into Date objects. This is important for consistent date handling across time zones.

```
projection = d3.geoAlbersUsa().scale(1280).translate([480, 300])
```

Sets up a `projection` function using D3's `geoAlbersUsa` method, which is a commonly used projection for creating maps of the United States. The projection is scaled and translated to fit the visualization's dimensions.

```
import {Scrubber} from "@mbostock/scrubber"
```

Allows users to interactively scrub (i.e., move through) a range of values (in this case, dates). It's likely used here to allow users to dynamically change the date being viewed in the visualization.

```
chart = {

  const svg = d3.create("svg")

            .attr("viewBox", [0, 0, 960, 600]);

  svg.append("path")

            .datum(topojson.merge(us, us.objects.lower48.geometries))

            .attr("fill", "#ddd")

            .attr("d", d3.geoPath());

  svg.append("path")

            .datum(topojson.mesh(us, us.objects.lower48, (a, b) => a !== b))

            .attr("fill", "none")

            .attr("stroke", "white")
```

```
    .attr("stroke-linejoin", "round")

    .attr("d", d3.geoPath());

const g = svg.append("g")

    .attr("fill", "none")

    .attr("stroke", "black");

const dot = g.selectAll("circle")

    .data(data)

    .join("circle")

    .attr("transform", d => `translate(${d})`);

svg.append("circle")

    .attr("fill", "blue")

    .attr("transform", `translate(${data[0]})`)

    .attr("r", 3);

let previousDate = -Infinity;

return Object.assign(svg.node(), {

    update(date) {

    dot // enter

    .filter(d => d.date > previousDate && d.date <= date)

    .transition().attr("r", 3);

    dot // exit
```

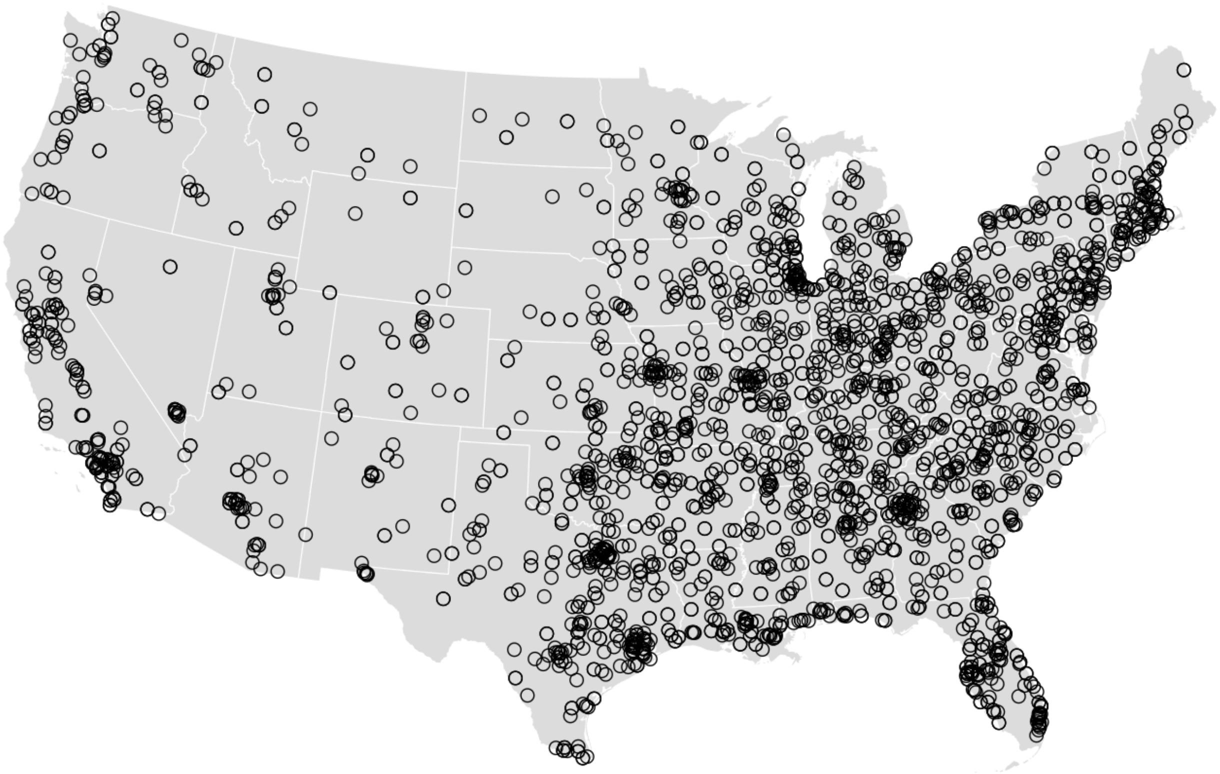
```
        .filter(d => d.date <= previousDate && d.date > date)

        .transition().attr("r", 0);

        previousDate = date;

        }

  });}
```

This line creates an SVG element using D3 and sets its view box to `960x600`. The view box is essentially the canvas area available for the SVG graphics.

Selects all `circle` elements within the group `g` (initially, there are none) and binds them to the data provided (`data`). The `join` method is used to handle the enter, update, and exit selections. Circles are positioned based on the data, likely representing points of interest or data points on the map, with the `translate` transform.

Question 2

Can you repeat the above work, but now you are aggregating the average location with the new Walmart outlet and the nearest one and plotting the new location with a circle?



https://observablehq.com/d/354acb87dc5260a4

The above is the link of the observable code and the transition in graph.

Code (d3.ls library of javascript)

```
chart = {

  // Assuming the existence of 'data', 'svg', 'topojson', and other necessary variables

  // Prepare the SVG container

  const svg = d3.create("svg")

            .attr("viewBox", [0, 0, 960, 600]);

  // Base map and outlines (assuming 'us' is defined appropriately)

  svg.append("path")

            .datum(topojson.merge(us, us.objects.lower48.geometries))

            .attr("fill", "#ddd")

            .attr("d", d3.geoPath());

  svg.append("path")

            .datum(topojson.mesh(us, us.objects.lower48, (a, b) => a !== b))

            .attr("fill", "none")

            .attr("stroke", "white")

            .attr("stroke-linejoin", "round")

            .attr("d", d3.geoPath());

  // Adjust the data structure

  const adjustedData = data.map(d => ({

coordinates: [d[0], d[1]],

date: new Date(d.date) // Ensure date is in Date object form
```

```
  }));

  /*const outletCircles = svg.append("g")

.attr("class", "outlets")

.selectAll("circle")

.data(adjustedData)

.join("circle")

        .attr("transform", d => `translate(${d.coordinates})`)

        .attr("r", 3)

        .attr("fill", "blue");*/

  const midpointGroup = svg.append("g");

  // Function to find the nearest outlet

  function findNearestOutlet(filteredData, targetOutlet) {

let nearestOutlet = null;

let shortestDistance = Infinity;

filteredData.forEach(outlet => {

        if (outlet === targetOutlet) return; // Skip comparing outlet to itself

        const distance = Math.sqrt(

        Math.pow(outlet.coordinates[0] - targetOutlet.coordinates[0], 2) +

        Math.pow(outlet.coordinates[1] - targetOutlet.coordinates[1], 2)

        );

        if (distance < shortestDistance) {
```

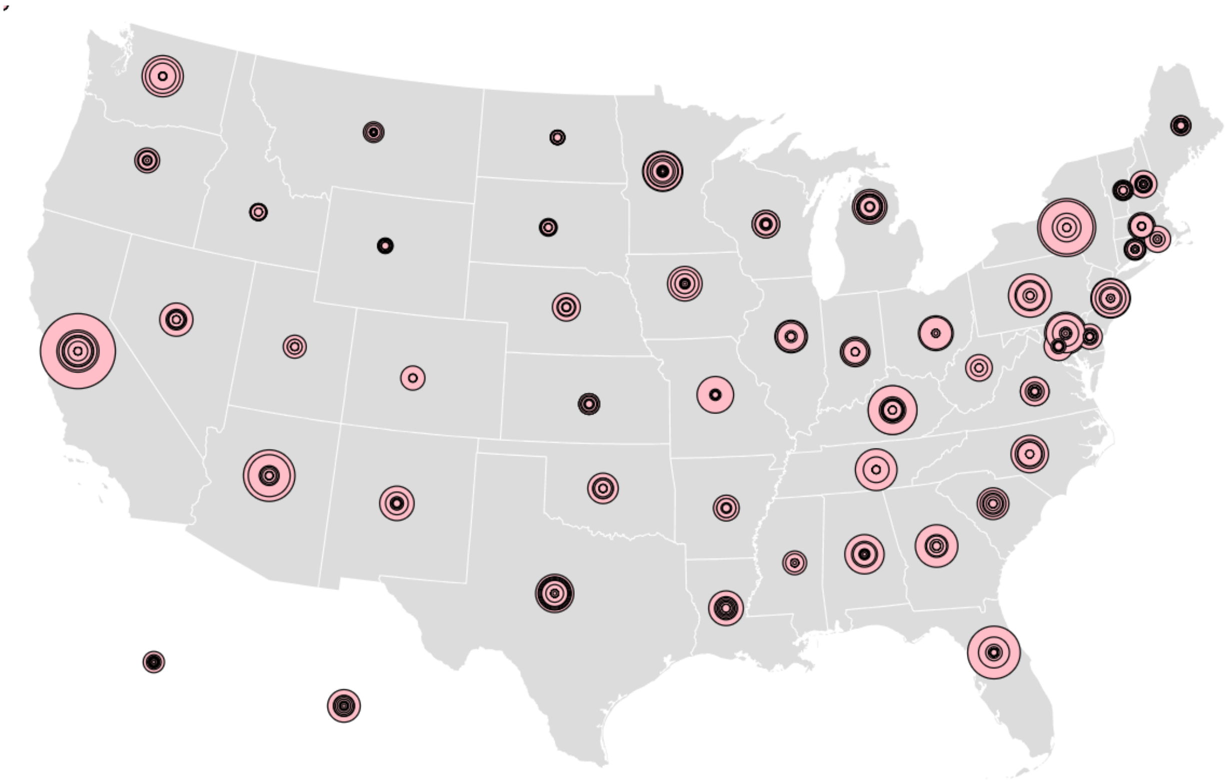
```
        shortestDistance = distance;

        nearestOutlet = outlet;

        }

});

return nearestOutlet;

  }

  // Function to calculate midpoints

  function calculateMidpoints(filteredData) {

return filteredData.map(outlet => {

        const nearestOutlet = findNearestOutlet(filteredData, outlet);

        if (!nearestOutlet) return null; // In case there's no nearest outlet

        const midpoint = [

        (outlet.coordinates[0] + nearestOutlet.coordinates[0]) / 2,

        (outlet.coordinates[1] + nearestOutlet.coordinates[1]) / 2,

        ];

        return { midpoint }; // Structure to include midpoint coordinates

}).filter(d => d !== null); // Filter out any null entries

  }

  // Update function to handle transitions and updates

  function update(selectedDate) {

const parsedSelectedDate = new Date(selectedDate);
const filteredData = adjustedData.filter(d => d.date <= parsedSelectedDate);
const midpointsData = calculateMidpoints(filteredData);
// Midpoint update logic
const midpoints = midpointGroup.selectAll("circle")
            .data(midpointsData, d => d.midpoint.join(","));
midpoints.enter().append("circle")
            .attr("transform", d => `translate(${d.midpoint})`)
            .attr("r", 0)
            .attr("fill", "None")
            .attr("stroke","black")
.transition().duration(500)
            .attr("r", 5);
midpoints.exit()
            .transition().duration(500)
            .attr("r", 0)
            .remove();
 }
 // Initialize with the latest date or another trigger mechanism
 const latestDate = d3.max(adjustedData, d => d.date);
 update(latestDate.toISOString());
 return Object.assign(svg.node(), { update });};
```

Question 3

Now consider the dataset ”U.S. Chronic Disease Indicators Dataset Link.” Solve the visualization problem and plot different diseases individually on the JavaScript library. The number of disease cases may be proportionate to the size of the circle in the plot.

https://observablehq.com/d/c69c5996a8f11395

The above is the link of the observable code and the transition in graph.



Code (d3.js library of javascript)

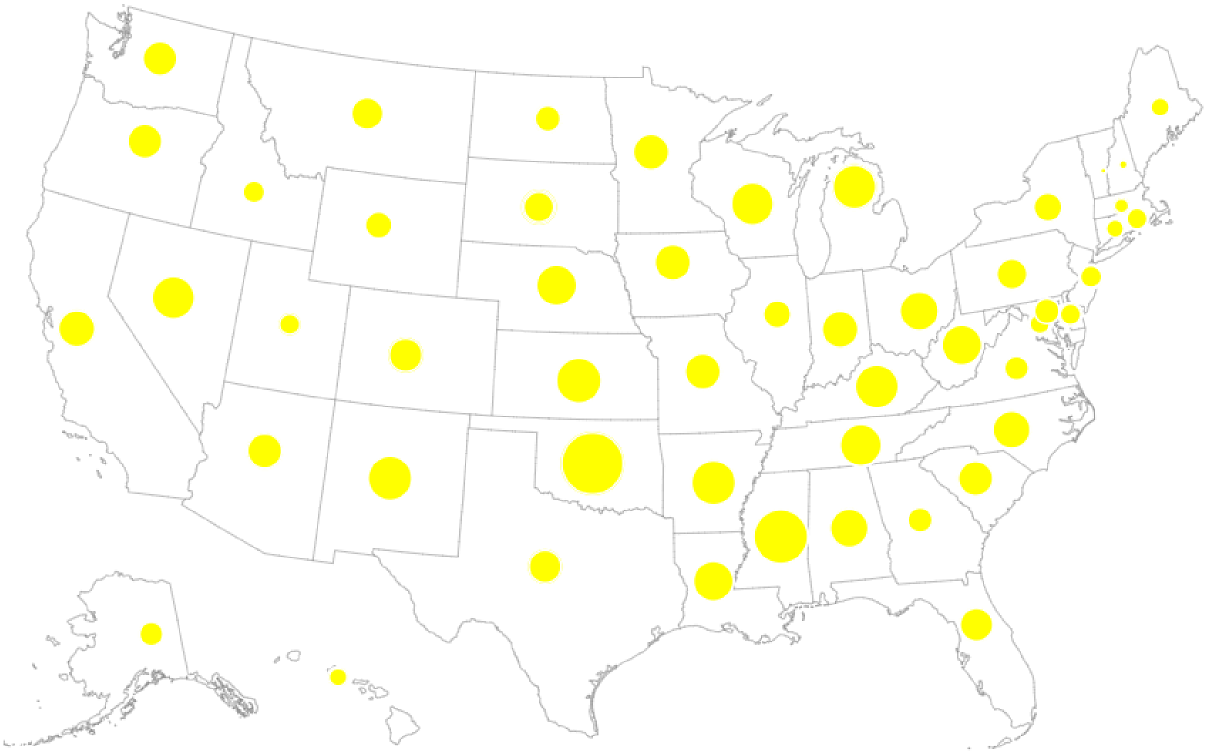
```
chart = {
  const svg = d3.create("svg")
      .attr("viewBox", [0, 0, 960, 600]);
  svg.append("path")
      .datum(topojson.merge(us, us.objects.lower48.geometries))
      .attr("fill", "#ddd")
      .attr("d", d3.geoPath());
  svg.append("path")
      .datum(topojson.mesh(us, us.objects.lower48, (a, b) => a !== b))
      .attr("fill", "none")
      .attr("stroke", "white")
      .attr("stroke-linejoin", "round")
      .attr("d", d3.geoPath());
  const g = svg.append("g")
      .attr("fill", "none")
      .attr("stroke", "black");
  const dot = g.selectAll("circle")
    .data(processedData)
     .join("circle")
      .attr("transform", d => `translate(${projection([d.longitude, d.latitude])})`)
      .attr("r", d=>Math.pow(d.cases,1/4))
      .attr("fill", "pink");
    let previousDate = 2000;
  return Object.assign(svg.node(), {
    update(date) {
      dot // enter
        .filter(d => d.year > previousDate && d.year <= date)
        .transition().attr("r", d=>Math.pow(d.cases,1/4));
      dot // exit
        .filter(d => d.year <= previousDate && d.year > date)
        .transition().attr("r", 0);
      previousDate = date;
    }
  });
}
processedData = data.reduce((accumulator, item) => {
  // Check for invalid data values
  if (item.DataValue === null || item.DataValue === "" || isNaN(+item.DataValue)) {
    return accumulator; // Make sure to return the accumulator as is
  }
  // Extract the longitude and latitude from the GeoLocation string
```

```
const pointRegex = /POINT \((-?\d+\.\d+) (-?\d+\.\d+)\)/;
const match = item.GeoLocation.match(pointRegex);
// Skip the item if there's no match for coordinates
if (!match) {
  return accumulator; // Make sure to return the accumulator as is
}
// Parse longitude and latitude from the matched groups
const longitude = parseFloat(match[1]);
const latitude = parseFloat(match[2]);
const year = +item.YearStart;
const state = item.LocationDesc;
const disease = item.Topic;
const cases = +item.DataValue;
// Create a unique key for each state, year, and disease combination
const key = `${state}-${year}-${disease}`;
let entryIndex = accumulator.findIndex(entry => entry.key === key);
// If an entry doesn't exist, create it
if (entryIndex === -1) {
  accumulator.push({
    cases:cases,
    key: key,
      state: state,

      year: year,

      diseases: { [disease]: cases },

      longitude: longitude,

      latitude: latitude

    });

  } else {

    // If an entry exists, update it

    let existingEntry = accumulator[entryIndex];

    existingEntry.diseases[disease] = (existingEntry.diseases[disease] || 0) + cases;

  }

  // It's crucial to return the accumulator for the next iteration

  return accumulator;

}, []); // Start with an empty array as the accumulator

update = chart.update(date)

JSZip = require("jszip@3.6/dist/jszip.min.js");

// Assuming JSZip is already loaded in your environment

data = FileAttachment("archive (5)@1.zip").arrayBuffer()

  .then(JSZip.loadAsync)

  .then(zip => zip.file("US_Chronic_Disease_Indicators.csv").async("string"))

  .then(d3.csvParse);
```

```
stateMesh = FileAttachment("us-counties-10m.json").json().then(us => topojson.mesh(us, us.objects.states))

parseDate = d3.utcParse("%Y")

projection = d3.geoAlbersUsa().scale(1280).translate([480, 300])

us = {
  const us = await d3.json("https://cdn.jsdelivr.net/npm/us-atlas@1/us/10m.json");
  us.objects.lower48 = {
    type: "GeometryCollection",
    geometries: us.objects.states.geometries.filter(d => d.id !== "02" && d.id !== "15")
  };
  return us;
}

import {Scrubber} from "@mbostock/scrubber"
```

Question 4

Now consider the dataset ”U.S. Chronic Disease Indicators Dataset Link.” Solve the problem of visualization and plot summation of different diseases on the JavaScript library.

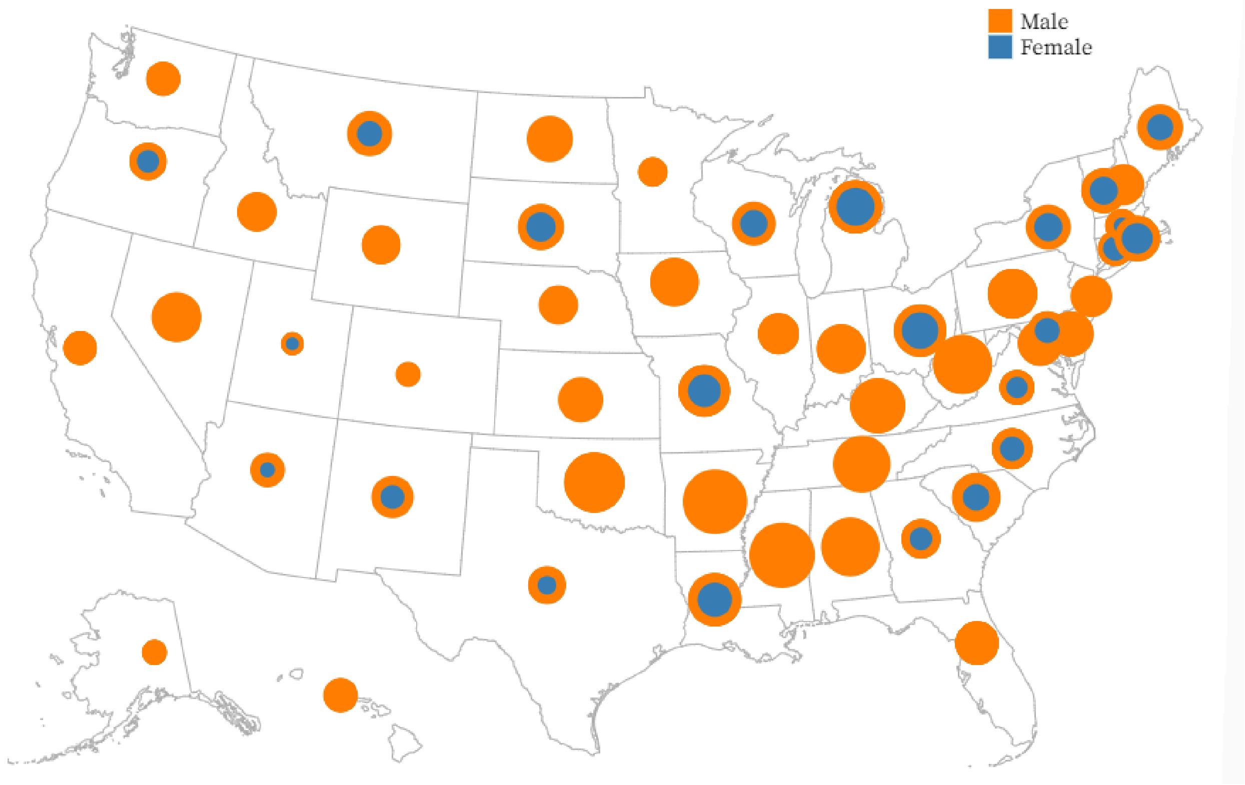


https://observablehq.com/d/3587e43f3feb285c

Link to the observable notebook for parts (d) and (e).

Question 5

There are parameters like StratificationCategoryID1 and StratificationCategory1. Use them in a meaningful way for the above task



https://observablehq.com/d/3587e43f3feb285c

Link to the observable notebook for parts (d) and (e).

Code (d3.js library of javascript)

```
data = FileAttachment("archive (5).zip").arrayBuffer()

  .then(JSZip.loadAsync)

  .then(zip => zip.file("US_Chronic_Disease_Indicators.csv").async("string"))

  .then(d3.csvParse);

JSZip = require('jszip@3.6.0/dist/jszip.min.js')

diseaseData = {};

processedData = data.reduce((accumulator, item) => {

  // Skip the item if the DataValue is null, not a number, or DataValueUnit is not "%"

  if (item.DataValue === null || item.DataValue === "" || isNaN(+item.DataValue) || item.DataValueUnit !== "cases per 100,000") {

    return accumulator;

  }

  const year = item.YearStart;

  const state = item.LocationDesc;

  const disease = item.Topic;

  const cases = +item.DataValue; // Convert DataValue to a number

  const coordsMatch = item.GeoLocation.match(/\(([^)]+)\)/);

  const coords = coordsMatch ? coordsMatch[1].split(' ') : null;

  // Skip the item if coordinates are not available

  if (!coords) return accumulator;
  const latitude = +coords[1];

  const longitude = +coords[0];

  // If the disease doesn't exist in the accumulator, add it

  if (!accumulator[disease]) {

    accumulator[disease] = {};

  }

  // If the year-state key doesn't exist for this disease, add it

  const yearStateKey = `${year}-${state}`;

  if (!accumulator[disease][yearStateKey]) {

    accumulator[disease][yearStateKey] = {

      year: year,

      state: state,

      cases: 0,

      latitude: latitude,

      longitude: longitude,

    };

  }

  // Aggregate the cases

  accumulator[disease][yearStateKey].cases += cases;

  return accumulator;

}, {});
```

```
aggregatedData = Object.entries(processedData).flatMap(([disease, yearStates]) => {
  return Object.entries(yearStates).map(([yearState, data]) => ({
    disease: disease,
    yearState: yearState,
    year: data.year,
    state: data.state,
    cases: data.cases,
    latitude: data.latitude,
    longitude: data.longitude
  }));
});
function filterDataByDisease(processedData, diseaseName) {
  // Check if the disease name is in the processed data
  if (!processedData.hasOwnProperty(diseaseName)) {
    console.warn(`No data found for disease: ${diseaseName}`);
    return [];
  }
  // Retrieve the year-state data for the specified disease
  const diseaseData = processedData[diseaseName];
  // Transform the year-state data into a flattened array suitable for plotting
  const filteredData = Object.entries(diseaseData).map(([yearState, data]) => ({
    disease: diseaseName,
    yearState: yearState,
    year: data.year,
    state: data.state,
    cases: data.cases,
    latitude: data.latitude,
    longitude: data.longitude
  }));
  return filteredData;
}
aggregatedArray = Object.values(processedData);
Chart3 = {
  const width = 928;
  const height = 581;
  const projection = d3.geoAlbersUsa().scale(4 / 3 * width).translate([width / 2, height / 2]);
  const svg = d3.create("svg")
    .attr("viewBox", [0, 0, width, height])
    .attr("width", width)
    .attr("height", height)
    .attr("style", "max-width: 100%; height: auto;");
  svg.append("path")
  .datum(stateMesh)
  .attr("fill", "none")
  .attr("stroke", "#777")
  .attr("stroke-width", 0.5)
  .attr("stroke-linejoin", "round")
  .attr("d", d3.geoPath(projection));
// Preprocess the data to sum up cases per state per year
let stateYearlyCases = {};
aggregatedArray.forEach(item => {
  Object.entries(item).forEach(([yearState, data]) => {
    if (!stateYearlyCases[yearState]) {
      stateYearlyCases[yearState] = {
        year: data.year,
        state: data.state,
        cases: 0,
        latitude: data.latitude,
        longitude: data.longitude
      };
    }
    stateYearlyCases[yearState].cases += data.cases;
  });
});

// Flatten the stateYearlyCases object into an array for D3

let flatData = Object.values(stateYearlyCases);

const maxCases = d3.max(flatData, d => d.cases);

const radiusScale = d3.scaleSqrt().domain([d3.min(flatData, d => d.cases), maxCases]).range([0, 25]);

// Use flatData to draw the circles

const circles = svg.selectAll("circle")

  .data(flatData, d => `${d.year}-${d.state}`)

  .enter().append("circle")

    .attr("transform", d => {

      const coords = projection([d.longitude, d.latitude]);

      return coords ? `translate(${coords})` : null;

    })

    .attr("r", d => radiusScale(d.cases))

    .attr("fill", "yellow")

    .attr("stroke", "white")

    .attr("stroke-width", 1.5)

    .append("title")

    .text(d => `${d.year}-${d.state}: ${d.cases} total cases`);

// Immediately set the radius of the circles based on cases without transition
```

```
  function update(year) {
    const yearData = flatData.filter(d => d.year === String(year));
    svg.selectAll("circle")
      .data(yearData, d => `${d.year}-${d.state}`)
      .attr("r", d => radiusScale(d.cases));
  }
  // Initial update to show the circles for the first year
  const initialYear = d3.min(flatData, d => d.year);
  update(initialYear);
  return Object.assign(svg.node(), { update });
};
processedData3 = data.reduce((accumulator, item) => {
  // Skip the item if the DataValue is null, not a number, or DataValueUnit is not "cases per
100,000"
  if (item.DataValue === null || item.DataValue === "" || isNaN(+item.DataValue) ||
item.DataValueUnit !== "cases per 100,000") {
    return accumulator;
  }
  // Skip the item if the stratification category is not "Gender"
  if (item.StratificationCategory1 !== "Gender") {
    return accumulator;
  }
// Skip the item if the disease is not "Diabetes"
if (item.Topic !== "Cardiovascular Disease") {
  return accumulator;
}
const year = item.YearStart;
const state = item.LocationDesc;
const disease = item.Topic;
const gender = item.Stratification1;
const cases = +item.DataValue; // Convert DataValue to a number
const coordsMatch = item.GeoLocation.match(/\(([^)]+)\)/);
const coords = coordsMatch ? coordsMatch[1].split(' ') : null;
// Skip the item if coordinates are not available
if (!coords) return accumulator;
const latitude = +coords[1];
const longitude = +coords[0];
// If the disease doesn't exist in the accumulator, add it
if (!accumulator[disease]) {
  accumulator[disease] = {};
}
// If the year-state-gender key doesn't exist for this disease, add it
const yearStateGenderKey = `${year}-${state}-${gender}`;
```

```
  if (!accumulator[disease][yearStateGenderKey]) {
    accumulator[disease][yearStateGenderKey] = {
      year: year,
      state: state,
      gender: gender,
      cases: 0,
      latitude: latitude,
      longitude: longitude,
    };
  }
  // Aggregate the cases
  accumulator[disease][yearStateGenderKey].cases += cases;
  return accumulator;
}, {});
Chart4 = {
  const width = 928;
  const height = 581;
  const projection = d3.geoAlbersUsa().scale(4 / 3 * width).translate([width / 2, height / 2]);
  const svg = d3.create("svg")
    .attr("viewBox", [0, 0, width, height])
    .attr("width", width)
  .attr("height", height)
  .attr("style", "max-width: 100%; height: auto;");
svg.append("path")
  .datum(stateMesh)
  .attr("fill", "none")
  .attr("stroke", "#aaa")
  .attr("stroke-width", 1)
  .attr("stroke-linejoin", "round")
  .attr("d", d3.geoPath(projection));
// Preprocess the data to sum up cases per state per year per gender
let stateYearlyGenderCases = {};
Object.entries(processedData3).forEach(([disease, diseaseData]) => {
  Object.entries(diseaseData).forEach(([yearStateGender, data]) => {
    const [year, state, gender] = yearStateGender.split("-");
    const key = `${year}-${state}-${gender}`;
    if (!stateYearlyGenderCases[key]) {
      stateYearlyGenderCases[key] = {
        year,
        state,
        gender,
        cases: 0,
      latitude: data.latitude,
      longitude: data.longitude,
     };
    }
    stateYearlyGenderCases[key].cases += data.cases;
   });
  });
  // Flatten the stateYearlyGenderCases object into an array for D3
  let flatData = Object.values(stateYearlyGenderCases);
  const maxCases = d3.max(flatData, d => d.cases);
  const radiusScale = d3.scaleSqrt().domain([d3.min(flatData, d => d.cases), maxCases]).range([0, 25]);
  // Define a color scale with distinct colors for male and female
  const colorScale = d3.scaleOrdinal()
   .domain(["Male", "Female"])
   .range(["#ff7f00", "#377eb8"]);
  // Use flatData to draw the circles
  const circles = svg.selectAll("circle")
   .data(flatData, d => `${d.year}-${d.state}-${d.gender}`)
   .enter().append("circle")
   .attr("transform", d => {
    const coords = projection([d.longitude, d.latitude]);
    return coords ? `translate(${coords})` : null;
   })
   .attr("r", d => radiusScale(d.cases))
   .attr("fill", d => colorScale(d.gender))
   // .attr("stroke", d => d.gender === "Male" ? "#377eb8" : "#ff7f00") // Darker stroke for better visibility
   // .attr("stroke-width", 3)
   .append("title")
   .text(d => `${d.year}-${d.state}-${d.gender}: ${d.cases} total cases`);
  // Add legend
  const legend = svg.selectAll(".legend")
   .data(colorScale.domain())
   .enter().append("g")
   .attr("class", "legend")
   .attr("transform", (d, i) => `translate(0,${i * 20})`);
  legend.append("rect")
   .attr("x", width - 180)
   .attr("width", 18)
   .attr("height", 18)
   .style("fill", colorScale);
  legend.append("text")
   .attr("x", width - 156)
   .attr("y", 9)
   .attr("dy", ".35em")
   .style("text-anchor", "start")
   .text(d => d);
  // Immediately set the radius of the circles based on cases without transition
  function update(year) {
   const yearData = flatData.filter(d => d.year === String(year));
   svg.selectAll("circle")
    .data(yearData, d => `${d.year}-${d.state}-${d.gender}`)
    .attr("r", d => radiusScale(d.cases));
  }
  // Initial update to show the circles for the first year
  const initialYear = d3.min(flatData, d => d.year);
  update(initialYear);
  return Object.assign(svg.node(), { update });};
stateMesh = FileAttachmen
t("us-counties-10m.json").json().then(us => topojson.mesh(us, us.objects.states))
import {legend} from "@d3/color-legend"
import {Scrubber} from "@mbostock/scrubber"
```