Basic Data Visualization with D3.js

Visualize data dynamically

Table of contents

Introduction

Meet John

John had CSV data tracking key metrics for his company that he wanted to visualize. Simply sharing the raw data wouldn't convey the insights effectively. He needed an interactive visualization solution.

John found D3.js, a JavaScript library to bind data to HTML/SVG and create visualizations. Using D3.js, John built line charts, bar charts and pie charts to represent his data.

John loaded his data into D3.js and created his first visualization in a few lines of code. He built more visualizations and dashboards, bringing his data to life with responsive and animated charts.

Stakeholders seeing John's dashboards instantly grasped insights that were hard to see in the CSVs. John's project succeeded, and he plans to use D3.js again. D3.js proved a simple yet powerful tool to gain valuable insights into complex data.

Overall, the core message and sequence of events remains the same as the original story, just in a more concise summarization.

What is D3.js?

- D3.js is a JavaScript library for manipulating documents based on data
- Allows you to bind data to a Document Object Model (DOM), and then apply transformations to the document
- Uses web standards: HTML, SVG and CSS
- Open source, maintained by Mike Bostock

Compare D3.js to other chart libraries

	D3.js	Highcharts	Vega	Vega Lite
Type	Library	Charting library	Visualization grammar	High-level visualization grammar
Language	JavaScript	JavaScript	JSON	JSON
Learning curve	Steep	Low	Moderate	Low
Customization	High	Medium	High	Low
Main purpose	DOM manipulation for visualization	Interactive charts and plots	Create custom visualizations	Rapid visualization creation

Compare D3.js to other chart libraries

Some key differences:

- D3.js has a steep learning curve as a low-level library, while the others are higher level (especially Vega Lite)
- D3.js and Vega offer high customization, while Highcharts and Vega Lite are more limited
- Highcharts is focused specifically on charting, while D3.js and Vega are more general purpose
- Vega and Vega Lite use a JSON grammar, while D3.js and Highcharts use JavaScript
- D3.js manipulates the DOM directly, while the others have abstraction layers Overall, the choice depends on your needs for customization vs. ease of use, purpose of visualization, and tech stack. Let me know if you would like me to clarify or expand the table further. I aimed for a high-level comparison, but can add more nuanced detail if needed.

Basic concepts

Basic Concepts

The core concepts to create charts using D3.js

There are some core concepts we need to know before creating our first chart using D3.js:

- Selections
- Scales
- Axes
- Transitions
- Data Binding

Selections

Selections are the basic unit of interaction in D3.js. They allow you to query and traverse the DOM.

Making a selection:

- Use d3.select() to select one element:

```
1   d3.select("div")  // Selects first div
```

- d3.select() uses querySelector() under the hood to find the first matching element.
- Use d3.selectAll() to select multiple elements:

```
1   d3.selectAll("div") // Selects all divs
```

- d3.selectAll() uses querySelectorAll() under the hood to find all matching elements.

Selections

Selections are the basic unit of interaction in D3.js. They allow you to query and traverse the DOM.

- Select by ID (#), class (.), tag name, attribute, or node:

```
d3.select("#id");   // Uses getElementById()
d3.select(".class");   // Uses getElementsByClassName()
d3.select("div");      // Uses querySelector()
d3.select("[attr]");  // Uses querySelector()
d3.select(node);      // Just passes through node
```

Selections

Selections are the basic unit of interaction in D3.js. They allow you to query and traverse the DOM.

Selecting and chaining:

- Store selection in a variable to operate on later
- Use method chaining to apply multiple operations to a selection sequentially:

```
let divSelection = d3.selectAll("div")
                     .attr("class", "bar")     // Set attribute
                     .style("color", "blue")   // Set style
```

- Each method call returns the selection, allowing the next method to be called on the same selection.

Selections

Selections are the basic unit of interaction in D3.js. They allow you to query and traverse the DOM.

Appending elements:

- Use .append() to add an element to each node in the selection:

```
1   divSelection.append("p")   // Add <p> to each div
```

- .append() uses element.appendChild() to append a child element to each node in the selection.

Selections

Selections are the basic unit of interaction in D3.js. They allow you to query and traverse the DOM.

Other selection methods:

- .attr() - Get or set attributes
- .text() - Get or set text content
- .html() - Get or set inner HTML
- .style() - Get or set styles
- .on() - Add event listeners
- .data() - Bind data (covered in Data Binding slides)
- .enter() - Get "enter" selection (covered in Data Binding slides)
- .exit() - Get "exit" selection (covered in Data Binding slides)
- .remove() - Remove elements from document
- And many more! Selections are a core part of D3.

Scales

Scales map a dimension of data to a visual representation. They are a core part of D3.js.

Why use scales?

- Map a continuous input domain (data) to a continuous or discrete output range (visual)
- Make visual encodings of data easier by handling interpolation and normalization
- Many types for different types of data (linear, ordinal, time, quantize, etc.)

Scales

Scales map a dimension of data to a visual representation. They are a core part of D3.js.

Creating a scale

- Use .scaleLinear() for a continuous linear scale:

```
let scale = d3.scaleLinear()
              .domain([0, 100])     // Data domain
              .range([0, 500])      // Visual range
```

- Use .scaleOrdinal() for a discrete ordinal scale:

```
let scale = d3.scaleOrdinal()
              .domain(["a", "b", "c"])
              .range(["red", "blue", "green"])
```

- Other scale types: time scale for dates, quantize scale for quantizing a continuous range into discrete values, threshold scale for a two-color range, etc.

Scales

Scales map a dimension of data to a visual representation. They are a core part of D3.js.

Scale functions:

- scale(x) - Get corresponding y value for x
- For example:

```
scale(25)     // Returns 250
scale("b")   // Returns "blue"
```

- invert(y) - Get x value for corresponding y value - domain() - Get or set input domain - range() - Get or set output range - ticks() - Get tick values for axis - tickFormat() - Format ticks appropriately (currency, percentage, time, etc.)

Scales

Scales map a dimension of data to a visual representation. They are a core part of D3.js.

Uses of scales:

- Position and size elements based on data
- Color elements
- Create axes to map positions visually
- Encode other visual properties like opacity, font size, etc.

Axes

Axes are visual representations of scales. They provide reference marks for values along a scale's domain.

Why use axes?

- Give viewers a reference to position, value, and units along a dimension
- Automate the generation of axes based on scales
- Control axis appearance by setting properties

Axes

Axes are visual representations of scales. They provide reference marks for values along a scale's domain.

Creating an axis:

- Call .axisBottom() (or .axisTop(), .axisLeft(), .axisRight()) and pass in a scale:

```
let scale = d3.scaleLinear()
              .domain([0, 100])
              .range([0, 500]);

let axis = d3.axisBottom(scale);
```

Axes

Axes are visual representations of scales. They provide reference marks for values along a scale's domain.

Appending the axis to the DOM:

- Create a g (group) element
- Apply a transform to position the axis
- Append the g to your SVG or grouping element

```
1   svg.append("g")
2      .attr("transform", "translate(0,500)")
3      .call(axis);
```

Axes

Axes are visual representations of scales. They provide reference marks for values along a scale's domain.

Axis properties: Some properties to customize your axis:

- ticks: Number of ticks (and labels)
- tickSize: Length of ticks
- tickPadding: Padding between tick and label
- tickFormat: Format tick labels ($, %, time, SI prefix, etc.)
- orient: Orientation (top, bottom, left, right)
- tickValues: Specific tick values to use instead of D3's default ** Uses of axes **
- x-axis along the bottom of a chart
- y-axis along the left side of a chart
- Color bar axes
- Size axes
- And more! Any visual encoding of data along a dimension.

Transitions

Transitions animate and interpolate between states in D3.js visualizations.

Why use transitions?

- Create engaging data visualizations
- Help the viewer understand changes between states
- Guide the viewer’s attention to important changes

Transitions

Transitions animate and interpolate between states in D3.js visualizations.

How do they work?

- D3 interpolates values and styles over the duration of the transition
- During a transition, D3 will calculate the state between the starting and ending values
  - For numeric values, it calculates the range between start and end
  - For colors, it interpolates RGB values
  - For positions, it calculates the intermediate x and y coordinates
- This gives the smooth animated transition effect

Transitions

Transitions animate and interpolate between states in D3.js visualizations.

Adding a transition: To a selection, call .transition() and chain a duration()method:

```
selection
  .transition()
  .duration(500) // Transition lasts 500ms
  .attr("width", 200) // End value
```

This will animate the width of elements in the selection from their current width to 200px over 500ms.

Transitions

Transitions animate and interpolate between states in D3.js visualizations.

Other methods:

- delay() - Delay transition start
- ease() - Set easing function for different effects (bounce, elastic, etc.)
- Can call same methods as on regular selections:
  - attr() - Animate attribute changes
  - style() - Animate style changes
  - text() - Transition text content
  - remove() - Transition elements out of the document

Transitions

Transitions animate and interpolate between states in D3.js visualizations.

Common uses:

- Change bar heights on updating data
- Zoom in/out of a map or scatterplot
- Fade elements in/out
- Move elements positions
- And much more! Transitions greatly enhance data visualizations.

Data Binding

Data binding links input data to elements in the document.

Why data binding?

- Dynamically create elements based on data
- Update, remove or modify elements based on updated data
- The foundation for interactive, data-driven visualizations in D3

Data Binding

Data binding links input data to elements in the document.

How does it work?

- Use .data() to bind data to a selection of elements
- There are three selections made:
  - Update selection: Existing elements with bound data
  - Enter selection: New elements bound to new data
  - Exit selection: Existing elements with no bound data
- Apply operations to each selection independently

Data Binding

Data binding links input data to elements in the document.

Update selection:

```
selection
  .data(data)  // Binds data to selection
  .text(d => d.value)  // Updates text content to match bound data
```

Enter selection: To append new elements for data not bound to an element yet:

```
selection
  .data(data)
  .enter()    // Selects enter selection
  .append("div")   // Append div for each enter selection
  .text(d => d.value) // Set text content with bound data
```

Data Binding

Data binding links input data to elements in the document.

Exit selection: To remove elements no longer bound to data:

```
selection
  .data(data)
  .exit()     // Selects exit selection
  .remove()   // Remove exit selection
```

Key functions: For stable data bindings, define a key function to uniquely identify elements:

```
selection
  .data(data, d => d.id)   // d.id is the key function
  // ...
```

Data Binding

Data binding links input data to elements in the document.

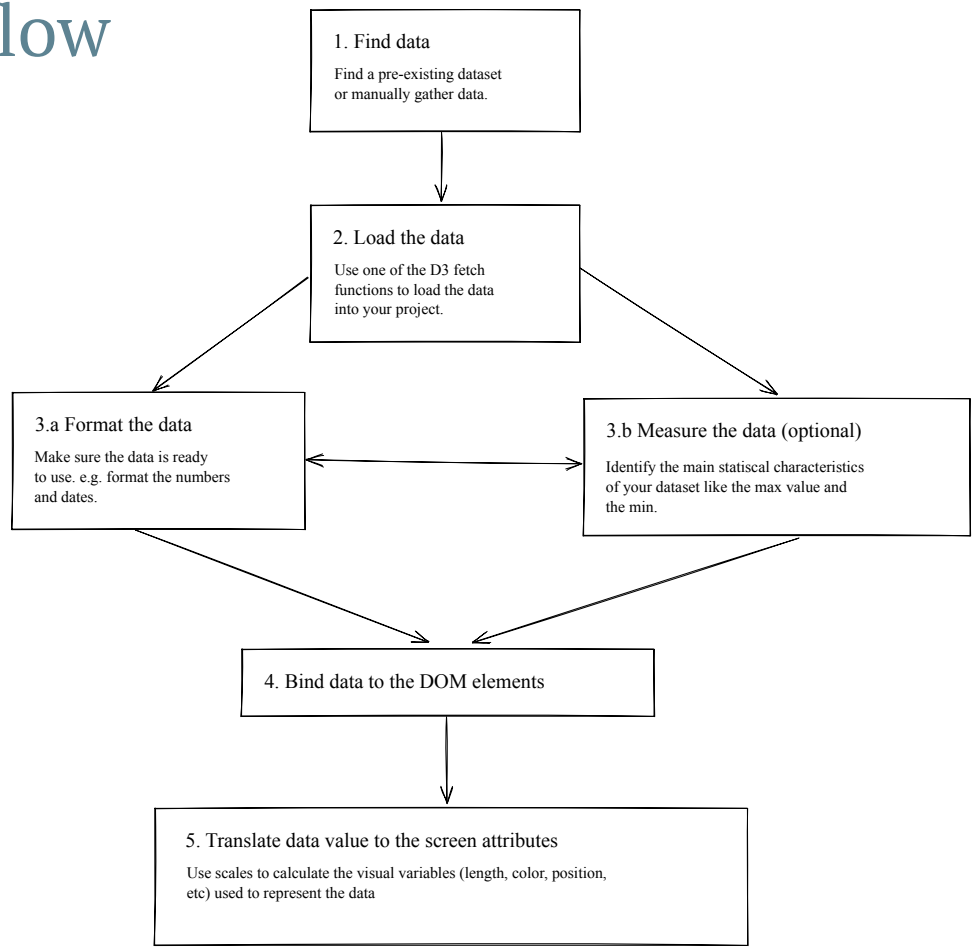
Chaining: Selections can be chained so you can bind data, update, add enter selections and remove exit selections sequentially.

Demo 1

Basic Concepts

Working with Data

The d3 data flow



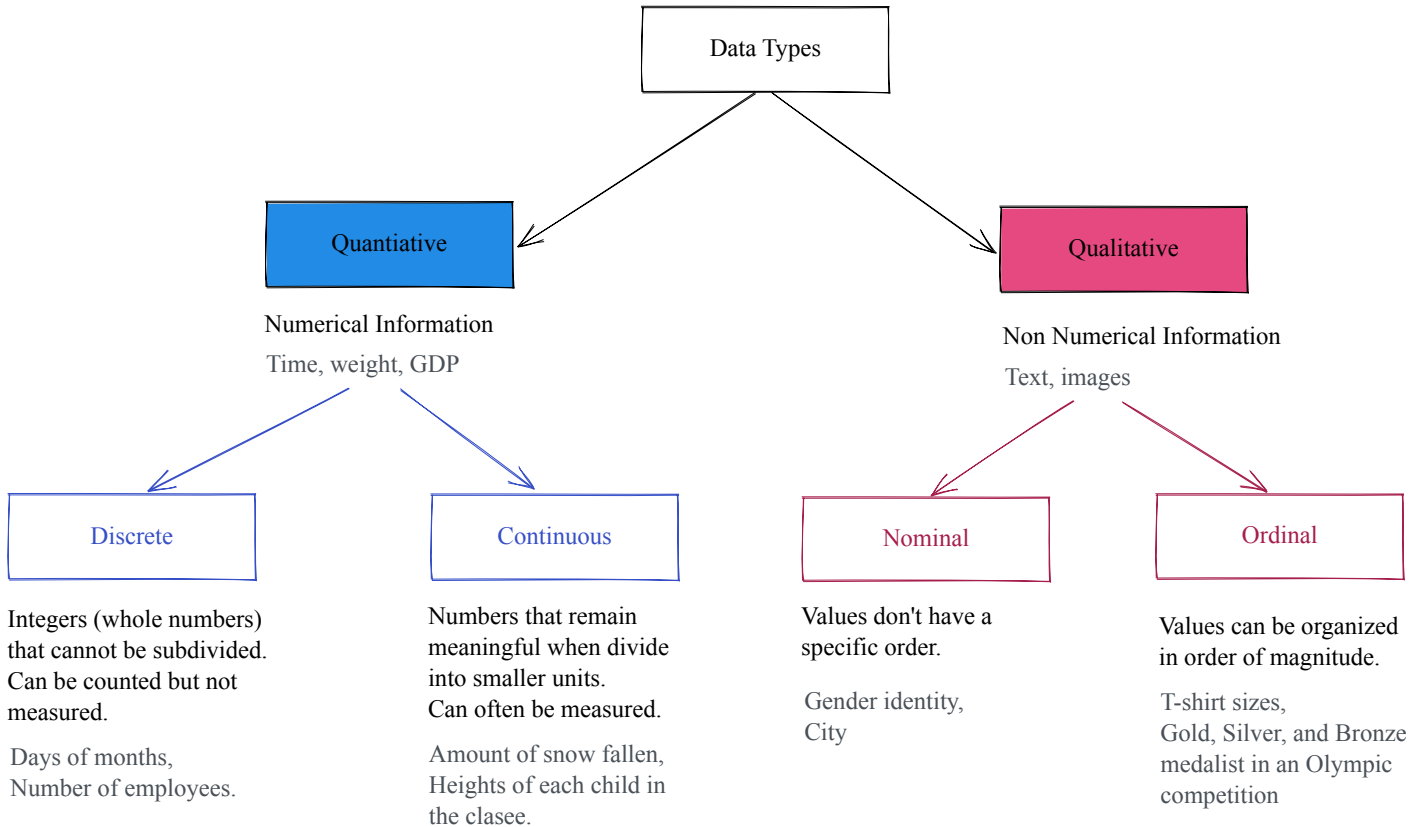
Find the data

Data Types

When building data visualizations, we work with two main data types: quantitative and qualitative. Quantitative data is numerical information like time, weight, or countries' GDP. Quantitative data can be discrete or continuous.

Discrete data consists of whole numbers, also called integers, that cannot be subdivided.

Qualitative data is made of non-numerical information like text. It can be nominal or ordinal. Nominal values don't have a specific order, for instance, gender identity labels or city names. Ordinal values, on the other hand, can be classified by order of magnitude. If we take t-shirt sizes as an example, we usually list them in ascending size order (XS, S, M, L, XL).


Data Types
Quantiative
Qualitative
Numerical Information
Time, weight, GDP
Non Numerical Information
Text, images
Discrete
Continuous
Nominal
Ordinal
Integers (whole numbers) that cannot be subdivided. Can be counted but not measured.
Days of months, Number of employees.
Numbers that remain meaningful when divide into smaller units. Can often be measured.
Amount of snow fallen, Heights of each child in the clasee.
Values don't have a specific order.
Gender identity, City
Values can be organized in order of magnitude.
T-shirt sizes, Gold, Silver, and Bronze medalist in an Olympic competition

Find the data

Data Formats and Structures

Data comes in many formats for different needs. Some common ones are:

- Tabular data: represent in columns and rows.
- JSON objects.
- Nested data.
- Networks.
- Geographic data.
- Raw data.

In this talk, we'll use the tabluar data from CSV file.

Load the data

We load the data into the project using the `d3-fetch` module.

```
d3.csv('src/data/1.csv', d => {
  console.log(d);
});
```

Format the Data

Note how the values from the count column have been fetched as strings instead of numbers. This is a common issue when importing data and is due to the type conversion of the dataset from CSV to JSON. Since the callback function of d3.csv() gives us access to the data one row at a time, it is a great place to convert the counts back into numbers. Doing so will ensure that the count values are ready to be used to generate our visualization later.

```
1  d3.csv('src/data/1.csv', d => {
2    console.log(d);
3    return {
4      technology: d.technology,
5      count: +d.count,
6    };
7  }).then(d => {
8    console.log(d);
9  });
```

Measure the Data

In step 3.a of our data workflow, we have completed the data formatting part but we can still explore and measure our data using D3. Measuring specific aspects of the data can help to get situated before diving into the actual crafting of a data visualization.

```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  console.log(data.length);
  console.log(d3.max(data, d => d.count)); // => 1078
  console.log(d3.min(data, d => d.count)); // => 20
  console.log(d3.extent(data, d => d.count)); // => [20, 1078]
  data.sort((a, b) => b.count - a.count);
  console.log(data); // => [20, 1078]
});
```

Bind the data to DOM elements

We are now ready to introduce one of the most exciting features of D3: data-binding. With data-binding, we can couple objects from a dataset to DOM elements. For instance, each rectangle element in our bar graph will be coupled with a technology and its corresponding count value. At the data-binding step of the data workflow, the visualization really starts to come to life.

To bind data, you only need to use the pattern shown in the next snippet and constituted of three methods (selectAll(), data() and join()) chained to a selection.

```
selection
 .selectAll("selector")
 .data(myData)
 .join("element to add");
```

Bind the data to DOM elements

Create the svg viewport

```
const svgWidth = 600;
const svgHeight = 700;
const selection = d3
  .select('.d3-content');
const svg = selection
  .append('svg')
  .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  .style('background-color', 'greenyellow');
```

Bind the data to DOM elements

Bind the data to DOM elements

```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  data.sort((a, b) => b.count - a.count);
  createViz(data);
});

function createViz(data) {
  const svgWidth = 600;
  const svgHeight = 700;
  const selection = d3
    .select('.d3-content');
  const svg = selection
    .append('svg')
    .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  const barHeight = 20;
  svg
    .selectAll('rect')
```

Bind the data to DOM elements

```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  data.sort((a, b) => b.count - a.count);
  createViz(data);
});

function createViz(data) {
  const svgWidth = 600;
  const svgHeight = 700;
  const selection = d3
    .select('.d3-content');
  const svg = selection
    .append('svg')
    .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  const barHeight = 20;
  svg
    .selectAll('rect')
```

Bind the data to DOM elements

```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  data.sort((a, b) => b.count - a.count);
  createViz(data);
});

function createViz(data) {
  const svgWidth = 600;
  const svgHeight = 700;
  const selection = d3
    .select('.d3-content');
  const svg = selection
    .append('svg')
    .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  const barHeight = 20;
  svg
    .selectAll('rect')
```

Bind the data to DOM elements

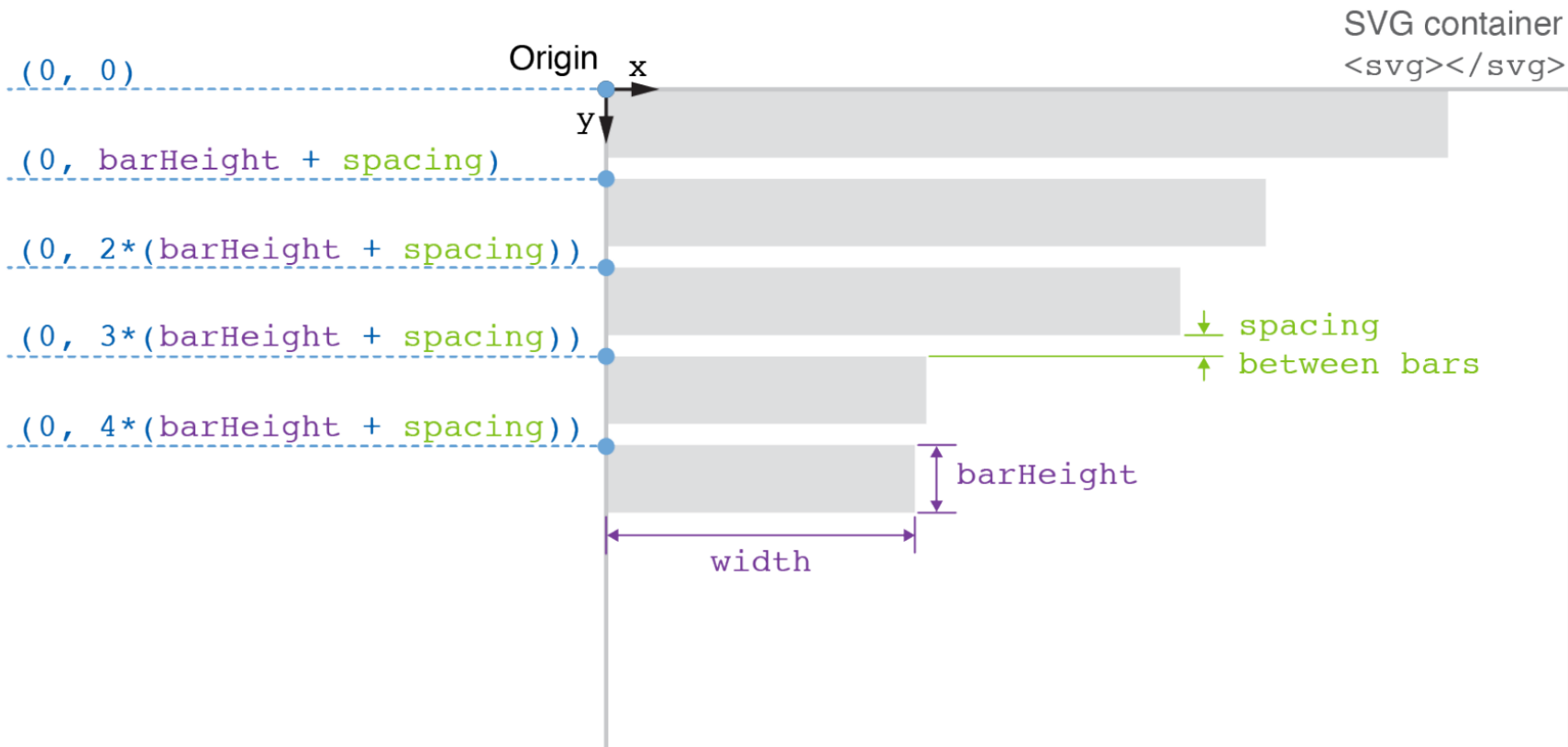
```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  data.sort((a, b) => b.count - a.count);
  createViz(data);
});

function createViz(data) {
  const svgWidth = 600;
  const svgHeight = 700;
  const selection = d3
    .select('.d3-content');
  const svg = selection
    .append('svg')
    .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  const barHeight = 20;
  svg
    .selectAll('rect')
```

Bind the data to DOM elements

```
d3.csv('src/data/1.csv', d => {
  return {
    technology: d.technology,
    count: +d.count,
  };
}).then(data => {
  data.sort((a, b) => b.count - a.count);
  createViz(data);
});

function createViz(data) {
  const svgWidth = 600;
  const svgHeight = 700;
  const selection = d3
    .select('.d3-content');
  const svg = selection
    .append('svg')
    .attr('viewBox', `0 0 ${svgWidth} ${svgHeight}`)
  const barHeight = 20;
  svg
    .selectAll('rect')
```


SVG container
<svg></svg>
Origin
x
y
(0, 0)
(0, barHeight + spacing)
(0, 2*(barHeight + spacing))
(0, 3*(barHeight + spacing))
(0, 4*(barHeight + spacing))
spacing
between bars
barHeight
width

Adapting the data for the screen

When we create data visualizations, we translate the data into visual variables, like the size of an element, its color or its position on the screen. In D3 projects, this translation is handled with scales.