Cold Assessment Tasks

Test 1)

Here below a possible solution for manipulating data of patients and calculate sensitivity and specificity of that data.

A link to test the code:

https://codepen.io/shiheb/pen/xodRwr

And here the code with some comments explaining the concept:

```
<!DOCTYPE html>
<html>
  <head>
     <meta charset="UTF-8">
     <script src="https://unpkg.com/react@16.7.0/umd/react.production.min.js"></script>
     <script src="https://unpkg.com/react-dom@16.7.0/umd/react-dom.production.min.js"></script>
     <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.24.0/babel.js"></script>
<style>
  span {font-weight: bold;font-size: 20px;}

</style>
  </head>
   <body>
    <div id="root"></div>

 <script type="text/babel">
const metadata = {
       patientA: "cancer",
```

```
    patientB: "cancer",
    patientC: "control",
    patientD: "cancer",
    patientE: "control"
};
const mutations = {
    patientA: {
        "KRAS/10394954": 5,
        "KRAS/3958838": 20,
        "TP53/94959003": 10,
        "TP53/12931920": 2,
        "NRAS/399322": 75,
    },
    patientB: {
        "KRAS/10394954": 31,
        "KRAS/3958838": 122,
        "TP53/94959003": 45,
        "TP53/12931920": 11,
        "NRAS/399322": 52,
    },
    patientC: {
        "KRAS/10394954": 10,
        "KRAS/3958838": 74,
        "TP53/94959003": 4,
        "TP53/12931920": 0,
        "NRAS/399322": 39,
        "NRAS/19929330": 20,
    },
    patientD: {
         "KRAS/10394954": 7,
         "KRAS/3958838": 3,
```

```
            "TP53/94959003": 11,
            "TP53/12931920": 92,
            "NRAS/399322": 0,
            "NRAS/19929330": 3,
           },
      patientE: {
            "KRAS/10394954": 7,
            "KRAS/3958838": 3,
            "TP53/94959003": 11,
            "TP53/12931920": 92,
            "NRAS/399322": 0,
            "NRAS/19929330": 3,
   }
};

// number of real positive cases in the data
var conditionPositive = 0;

//number of real negative cases in the data
var conditionNegative = 0;

// mutations_total will contain the total number of mutation per patient
var mutations_total = [];

function SommeMutation(mutations_total){
var i = 0;

for (let [keyy, valuee] of Object.entries(mutations)) {

var obj2 = Object.entries(mutations)[i][1];
```

```
var sum = 0;
for (let [keyyy, valueee] of Object.entries(obj2)) {

sum = sum + valueee;

}
i++;
mutations_total.push([sum]);

}
return (mutations_total);
}

function NumRealPos(){

for (let [key, value] of Object.entries(metadata)) {

if (({value}.value)=="cancer"){

  conditionPositive++;
}
}
return(conditionPositive);
}

function rounding(x) {
 return Number.parseFloat(x).toFixed(2);
}

mutations_total = SommeMutation(mutations_total);
```

```
// make a copy of our table before sorting it
const unsortedMutations = [...mutations_total];

// sort the table from left to right
mutations_total.sort(function(a, b){return b - a});



conditionPositive= NumRealPos();

conditionNegative = mutations_total.length -conditionPositive ;

var obj3= Object.values(metadata);

var j;
var truePositive=0;   //Sick people correctly identified as sick
for (j = 0; j < conditionPositive; j++) {

let ids = unsortedMutations.indexOf(mutations_total[j]);
if (obj3[ids]=="cancer")
{truePositive++;}
}

var k;
var trueNegative=0;  //Healthy people correctly identified as healthy
for (k = mutations_total.length-1 ; k >mutations_total.length - conditionPositive ; k--) {

let idx = unsortedMutations.indexOf(mutations_total[k]);
if (obj3[idx]=="control")
{trueNegative++;}

}
```

```
var sensitivity = truePositive/conditionPositive ;

var specificity = trueNegative/conditionNegative ;


 function Dataset(){


       return (

          <div>

              <p>The total number of people in this set of data is  <span> {mutations_total.length} </span> </p>

              <p>The number of sick people correctly identified as sick of this set of data is  <span> {truePositive} </span> </p>

              <p>The number of healthy people correctly identified as healthy of this set of data is <span> {trueNegative} </span> </p>

              <p>The Sensitivity of this set of data is  <span> {rounding(sensitivity)} </span></p>

              <p>The Specificity of this set of data is  <span> {rounding(specificity)} </span></p>

           </div>

       );

    }


   ReactDOM.render(

    <Dataset/>,

    document.getElementById('root')

  )

</script>

 </body>

</html>
```

Test 2)

In order to resolve the problem and try to find out a protentional approach that can help us to overcome this issue. Let's understand and point some limitations of what is already did to perform this situation. The rendering engine is single threaded. Almost everything, except network operations, happens in a single thread.

We can Use the Chrome DevTools Timeline panel to record and analyse all the activity in our application as it runs. It's the best place to start investigating perceived performance issues in our application and identify exactly where jank is happening and then destroy it.

In our case, rendering the preview of the table sticks and janks. To find out what the cause is, trying to scroll the screen to view the below the fold part while recording with chrome Dev tools timeline. It's important to realise that when scrolling, the device is going to put up a new picture or frame onto the screen for the user to see most devices today refresh their screen 60 times a second which we measure in Hertz and so to match that we need to have 60 frames that put up most of the time. If in any case this frame rate drops, we can easily spot in jank invaders since the browser is taking too long to make a frame it will get missed out the frame rate will drop and users will be seen stuttering.

The (FPS: Frames Per Second) graph indicate long frames, which are likely candidates for jank.

One of the nice features of tables is that one can let the browser handle the width of table cells. it will re-render the table with an increased width for the specific column. So, to escape the Layout part, one of the five major areas in the pixel pipeline, one can speed up the rendering by setting the table's CSS property 'table-layout' to 'fixed'. so, the browser can reserve a pre-defined layout slot early to avoid layout jumps during the page load. The Web Workers API allows us to create worker threads that can be used to execute code in the background so that the main thread is free to continue its execution, possibly processing UI events, ensuring no UI freeze

Moreover, we know that there is a waiting period for the component to actually render on the screen, and then it renders all at once. we want is to be able to show the items already loaded in virtual DOM, while we wait for others to load. To get this done, we need to render our ParentComponent as soon as possible above the fold (make it visible on the screen), and then add items in steps.

Instead of deserializing the array in chunks, the simplest way to refactor the program (code-wise) using multiple threads to shorten the amount of time it takes to render a new chunk of data. As the Multi-threaded optimization does not affect 1. Reading data from the table 2. Preparing and retrieving data from the database. 3. Randomizing and creating chunks. 4. Writing to the table, which run serially. One of the most efficient solution to improve performance and optimization of rendering of the data in the browser is to run an independent script in background, unbound to the user-interface scripts (isolated from the web page) utilizing with the most effectively way CPUs with their multi-core.

Web worker is tool (JavaScript script) especially developed for that purpose. Enabling multi-threading on the front end by spawning new background threads and running scripts in isolation, Web workers do not interfere with the main thread and consequently do not interrupt the user-interface, which will necessary resolve the problem of marshaling the huge amount of data to transfer it between the worker thread and the main thread.

In order for the worker to handle these messages, it must implement handler functions that deal with the message event. Passing data to a worker requires us to serialize the table and clone it by using the structured cloning algorithm or by using transferable objects.

In Case of structured cloning algorithm and transferable objects still inoffensive in front of this issue, we need a new concept that helps us to not having the trouble of passing updated data between threads and all threads can access and update the same data.

Shared memory is, for now, our last resort. Instead of passing the data copies between threads, we pass copies of the SharedArrayBuffer object. A SharedArrayBuffer object points to the memory where the data is saved. After creating the buffer of the shared memory, web workers store values in the shared memory. The Main thread will bring the serialized table from its place in the buffer with SharedArrayBuffer.prototype.slice(begin, end).

All what we have to manipulate is the cost of the shared memory. In other words, to preserve the frame rate of 60, rendering of one frame must not exceed ~12 ms, since 16.66 ms is the maximum that can take a frame to render to keep the rate.

References:

1. https://itnext.io/handling-large-lists-and-tables-in-react-238397854625
2. https://developers.google.com/web/fundamentals/performance/rendering/
3. https://classroom.udacity.com/courses/ud860/lessons/4138328558/concepts/41368485780923
4. https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/#Optimizations
5. https://stackoverflow.com/questions/20870448/reactjs-modeling-bi-directional-infinite-scrolling
6. https://stackoverflow.com/questions/21238667/infinite-scrolling-with-react-js/21240835#21240835
7. https://github.com/bvaughn/react-window
8. https://github.com/seatgeek/react-infinite
9. https://bvaughn.github.io/react-virtualized/#/components/CellMeasurer
10. https://bvaughn.github.io/react-virtualized/#/components/List
11. https://stackoverflow.com/questions/42159728/react-dont-render-components-in-table-who-arent-visible
12. https://mobx.js.org/best/react-performance.html
13. https://css-tricks.com/rendering-lists-using-react-virtualized/
14. https://www.keycdn.com/blog/rail-performance-model
15. https://developers.google.com/web/fundamentals/performance/rail
16. https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/
17. https://medium.com/tech-tajawal/threading-in-nodejs-5d966a3b9858
18. https://www.html5rocks.com/en/tutorials/workers/basics/
19. https://dev.opera.com/articles/web-workers-rise-up/
20. https://www.hongkiat.com/blog/shared-memory-in-javascript/
21. https://50linesofco.de/post/2017-02-06-javascript-in-parallel-web-workers-transferables-and-sharedarraybuffer
22. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer