Performance Optimization Techniques for ReactJS

Arshad Javeed
arshadjaveed171@gmail.com

***Abstract*— React is one of the popular web frameworks that has gained importance over other frameworks such as Angular, Vue, etc.. This is because of its implementation of Virtual DOM, whose primary objective is to enhance the overall performance of the application.**
However, there are certain things that one has to keep in mind before designing the applications. Failing to anticipate the problems that may occur component hierarchy will lead to performance degradation. Some of the commonly faced problems are component re-rendering, application lag due to background computations being run, lag due to processing large data sets in a single stretch, etc.

This paper will describe some of the practical ways of overcoming such problems within the application, thus enhancing the performance of the ReactJS App in a production environment. The paper will also describe a time-efficient search algorithm that can be used for searching objects in a large data set.

Keywords*— ReactJS, Performance, stability, multithreading, JavaScript, re-render, Speed, Search

I. Introduction

React is basically a web framework that was mainly designed to address the performance issues in the web application. React uses virtual DOM that decides whether the component has to be reloaded or not based on the current state of the component and the changes that have occurred. This prevents the application from re-rendering unnecessarily. Apart from this React also introduces one-way data flow which helps to control the flow of the data within the application which makes the tracking of the occurred easier and also simplifies the propagation and the stability.

The states and the properties (props) of the component are the two parameters that determine when the component should re-render in the application. Whenever there is a change or when a parent passes a new property to the child, the React DOM compares the new values to the previously stored values and only re-renders only if there is a difference between the two states.

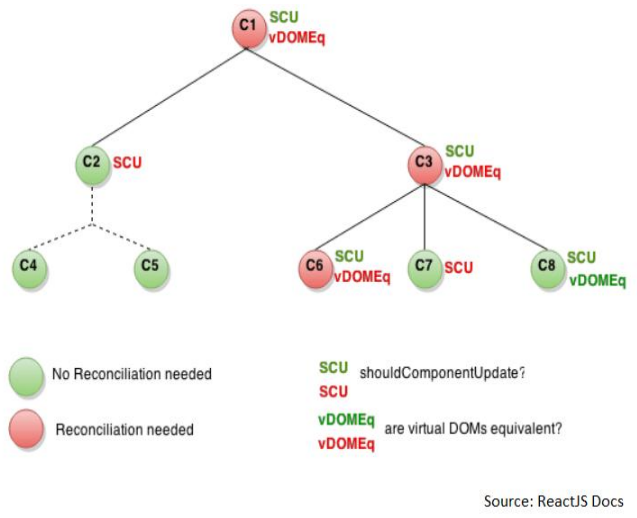

Figure 1: Component Tree highlighting the cost of rendering a React Component

Consider the component hierarchy as shown in Fig 1.
Let's say that due to some change in the component C1, the component C1 decided to re-render. Now React goes on checking the children in the subtrees of this component in a recursive manner until all the components in the subtree have been updated based on the value returned by the shouldComponentUpdate() method.
It is important to note that the components in the subtree are not forced to re-render by the parent component and instead the re-rendering of the parent component only means that the children in that subtree will be evaluated.
From the figure, it is clear that, since SCU of C1 is true, the subtrees C2 and C3 are checked. Since SCU of C2 is false, its children are not evaluated, whereas the components in the other subtree C3 are evaluated since SCU for C3 is true and similarly the process is continued. Finally, the components C1, C3, and C6 are re-rendered.

This type of re-rendering may not cause any issues in the case of a smaller application, but in complex applications where there are a huge number of components, this re-rendering causes serious performance issues.

And relying only on the React Virtual DOM comparison may not be sufficient. Instead, some additional measures have to be taken to re-render the components only when it is required.

The subsequent sections describe some ways of improving the performance of the React Application.

II. Optimizing the React App Performance

A. *Reducing the number of State and Prop variables*

This is an obvious measure to be taken. By reducing the number of State and Prop variables, the chances of the unnecessary re-renders of the component can be reduced.
Also avoiding frequent and unnecessary changes to the State and the Props can be helpful. The state has to updated immediately only if has a visual impact on the user. Else the state has to be updated only at the end.
Another thing that can be done is to prevent the component from rendering is to wait until all of the essential data has been received by the component. This can be done by overriding the shouldComponentUpdate(props, state) method of the component. This method is responsible for deciding whether the component should render or not.

Consider a scenario where a component will receive the data by making a REST API call to the server and also some local props received from other components within the application. And the component is rendered when it receives the server. If the data from the server is received first and then the props from other components are received, then this would cause re-render. Instead, adding a check in the shouldComponentUpdate() to validate the data can avoid the problem.

B. *Splitting the Main Component into Independent Components*

This is an important step that helps to prevent re-rendering of unnecessary DOM elements in the component.

Consider the following code snippet in which the component is just rendering a table where clicking on any particular row would set that particular element to the state.

```
render() {
    let data = [{name:abc},{name:xyz}, ...]
    return (
    . . .
    <Table>
      <Table.Header>
        <Table.HeaderCell>
          Name
        </Table.HeaderCell>
      </Table.Header>
      <Table.Body>
       {data.map(user =>
        <Table.Cell onClick={(user) =>
        {this.setToState(user)}}>
        {user.name}
         </Table.Cell>)}
      </Table.Body>
    </Table>
    . . .
    )
  }
```

At first glance, it would seem that there is nothing wrong with the above code. But as mentioned earlier, the component would re-render every time the state of the component changes. This means that every time the cell is clicked, the entire table is forced to re-render which causes performance issues. Of course, this is negligible if the size of the data is small (say about 100 or so). But in real time applications, the size of the data can be huge (in millions). And rendering the entire table on every click is very inefficient.

A better way is to isolate each table row as an independent component and enable these components to listen and act on events independently. By doing so, every time a click event occurs, only the one particular row will have to be re-rendered instead of re-rendering the entire table.

This is shown in the following code snippets

a. Main Component

This component is only responsible for creating and managing the children and does not listen to all the events of the children unless the child passes a prop to the parent.

```
render() {
    let data = [{name:abc},{name:xyz}, ...]
    return (
    . . .
    <Table>
      <Table.Header><Table.HeaderCell>Name</T
able.HeaderCell></Table.Header>
      <Table.Body>
       {data.map(user => <Row user={user}/>)}
      </Table.Body>
    </Table>
    . . .
    )
  }
```

b. Sub Component

This is an independent component, that is capable of handling all the events that occur, without having to notify other components.

```
  render() {
    return (
      <Table.Row>
        <Table.Cell
         onClick={this.setToState}>
         {this.props.user.name}
        </Table.Cell>
      </Table.Row>
    )
  }
```

Here each Row component will have its own state and every time there is a change, react virtual DOM will only compare the state for one particular row and re-render it if required.

This is unlike the previous case, where the entire table was re-rendered.

Essentially, the parent is just creating the instances and manages their life-cycle while the children operate individually without affecting the other siblings in the hierarchy.

Since React provides only one-way data binding, an additional event is required in the child element to publish the result back to the parent.

C. *Utilize Existing Component Instances*

Considering the previous example, let's say that the size of the data set is in the order of millions. Creating all the instances at once will take a lot of time and slow down the application.

Hence in order to avoid this, we can create a fixed number of instances and re-render them with the different Props whenever required.

In this case, we can create about 1000 instances initially and then, whenever the user makes a request, we pass the references of the next 1000 objects to the existing component instances. It is important to pass only the references to the object instances, as performing a deep copy of the objects might result in duplicating the instances and wastage of memory.

D. *Search Optimization*

Searching the large data sets involving millions of objects can be time-consuming. If the data received from the server is in no particular order, then the time complexity will be O(n). Also creating a single huge data structure of JSON objects will cause a problem with memory issues. In fact, trying to build the application with an array of Objects with the length close to about 100,000 will cause the build to fail (this again depends on the size of each object in the array). The Node Package Manager (npm) throws Heap out of memory error.

To solve this problem, we can split the data into a number of chunks of fixed sizes and at the same time use hashing, with the search attribute as the key.

Suppose the data elements are:

```
{name: abc, age: xx, ...}
{name: xyz, age: xx, ...}
.
.
.
{name: user1, age: xx, ...}
{name: user2, age: xx, ...}
.
.
.
```

Then, we will have to create a data structure as follows:

```
hash("abc") -> {name: abc, age: xx, ...}
hash("xyz") -> {name: xyz, age: xx, ...}
.
.
.
hash("user1") -> {name: user1, age: xx, ...}
hash("user2") -> {name: user2, age: xx, ...}
.
.
.
```

Here the Hash generated using the name of each element is used as the key to point to the Object. Hence whenever there is a necessity to search the object based on the name, the Hash for the corresponding string can be computed and can be used to directly find the object. The time complexity for the search will be O (1).

It is important to note that since we have used hashing, if the data is searched with the query string "user", then no results will be found. This is because Hash("user") and Hash("user1") will be different. For this reason, we need to compute some more hashes by building the search string further.

For example, if the data consists of the string "user", then it is more likely that strings such as "user1", "user2", "userxx", etc. may also be present in the data set. Hence for the search string "user" we can build such strings and also try to look up the corresponding Hashes in the Hashtable.

The time complexity in doing such operations will be k * O (1). Where k is the number of combinations of the search string. In comparison to the O (n) or even O (n log n) approach, this will provide better performance.

Suppose the data set consists of the following strings as an Object attribute:

"user", "user1", "user2", . . ., "user22", . . .

And let the query string be "user".

We can directly retrieve the object corresponding to the hash("user"), but while performing a search in real time, we are often interested in the related data as well. i.e. we would also expect the search to get the objects corresponding to strings, “user1”, “user2”. But this is not possible with simple Hashing.

This is where we can build similar strings and compute the hashes and use them to quickly look up the objects. So in this case, we may construct the following strings:

“user1”, “user2”, . . ., “userxx”

And look up the HashTable using these strings one at a time. Whenever a particular string, say “userx” is not found to be in the HashTable, we can switch to building strings in a new form – “userxx”.

Similarly, we can also build a few other string combinations as well. The total number of combinations built (k) determines the complexity of the search algorithm. In a sequential search, the time complexity will O (10^6) (assuming the length of the data set to be 1 million) whereas in this case, the time complexity will be k * O (1). Where k depends on the search string.
This is more of an absolute search, where the data that highly resembles the search query is fetched, rather than fetching all the data even with a slight resemblance. This is the tradeoff made to reduce the computation time.

But the main drawback is, having to fix the search attribute. Since we are building the Hash Table using a specific object attribute, it is not possible to change it unless we regenerate HashTable using a different attribute. But if the search attribute is fixed, then this approach seems promising.

E. Multithreading

Usually, the web browser spawns one thread per tab opened and this single thread will be responsible for all the operations that are performed in the application. This means if there are a large number of computations to be performed then, the thread would have to stop all other operations such as DOM manipulation, animation, rendering, etc. And the application will not be responsive during this time.

But in recent days, Google and Mozilla have introduced Web workers to make the browsers more powerful.
A web worker is a JavaScript program that runs on a different thread in parallel with the main thread. This enables us to implement multithreading in the application and perform parallel execution.

Some of the operations that can be performed are:

- Data and web page caching
- Image manipulation and encoding (base64 conversion)
- Canvas drawing and image filtering
- Network polling and web sockets
- Background I/O operations
- Video/Audio buffering and analysis
- Virtual DOM diffing
- Local database operations
- Computationally intensive data operations

Consider a scenario where there is a necessity to filter the data in our application. If the size of the data is small, then there is no need to have multiple threads running. But if the size of the data is very large, then it will not be efficient for a single thread to filter the entire data. In such cases, we can create multiple web workers and assign each web worker a specific portion of the entire data. And each worker would filter the portion of data assigned to it and store it in some specific place. After all the web workers complete the task, the main application thread can just merge the pieces of the filtered data.

Similarly, these web workers can also be used to perform other background and time-consuming tasks, while the application still functions perfectly without any performance degradation.

It is important to note that every time a web worker is created, it takes away some part of the system resources and these web workers are “killed” when the main web worker (thread) dies. So one should not overkill by creating a large number of web workers. Browsers like Firefox support up to 512 web workers to run simultaneously.

III. Conclusion

React is a great framework having its own way to tackle the performance issues that the Web Applications commonly face. But let alone, the performance can still be degraded when designing complex applications where the application has to deal with a lot of data processing. This leads to the application not being responsive and application lag. This is a common problem in Enterprise Applications.
This paper proposes some practical ways of tackling such problems within the web application.

The methods described aim to eliminate the redundant computations and optimize the performance of the application.

While some of the techniques described are specific to the React framework, other techniques like search optimization and multithreading are related to the performance of the application itself and can be implemented in any of the application, irrespective of the framework.

References

[1] "Optimizing Performance," ReactJS Docs, reactjs.org/docs

[2] C. Wohlie et al., "Experimenting in Software Engineering," Springer, 2012

[3] S. Frostl, "AngularJS Performance Tunning for Long Lists," blog, 2013, tech.small-imporvements.com

[4] Noam Elboim, "How to greatly improve your React App performance," blog, medium.com

[5] Yu Yao, Jie Xia, "Analysis and Research on the performance Optimization of Web Application system in high Concurrency Environment", IEEE, 2016

[6] "Web Workers API," MDN Docs, developer.mozilla.com.

[7] Darrel Greenhill, Jack Francik, Jay Kiruthika, Souheil Khaddaj, "UX Design in Web Applications," IEEE, 2016

[8] Sanjeev Dhavan, "Analysing Performance of Web-Based Metrics for Evaluating Reliability and Maintainability," IEEE, 2008