Tending to the Restrictions of React JS

Swarup kumar Gupta[1], Dr. Deepika K.[2]
PG Student, Department of MCA, RV College of Engineering, Bangalore, India[1]
Assistant Professor, Department of MCA, RV College of Engineering, Bangalore, India[2]

ABSTRACT

React is an internet framework that has higher options compared to alternative similar frameworks like Angular, Vue.JS. This is often because of its implementation of the Virtual DOM; whose goal is to enhance the overall capability of the application. In any case, there are a few things that should be remembered before designing the applications which if ignored, then issues may lead to execution issues. A portion of the normally confronted issues are element re-rendering, lag because of a process of massive data sets during a single stretch, and application lag because of background computations being run, etc. This paper provides alternative ways to overcome such problems, so enhancing the performance of React during a production atmosphere.

KEYWORDS: React, State, Props, Components, Multi-threading, Optimization, Performance, JavaScript

I. INTRODUCTION

React is a web framework that was designed to address the performance issues in a web application. It uses virtual DOM which decides if the component must be reloaded based on the current state of the component and any changes that occur. This helps to prevent the application from re-rendering if not required. React has a two-way data flow that controls the flow of the data inside the application which makes tracking easier and improves propagation and stability

The props and states of the component are two parameters that determine when a component should re-render in the application [1]. When there is a change or when a parent passes a new property to the child, the React DOM compares the new values to the previous values and re-renders only if there is a difference.

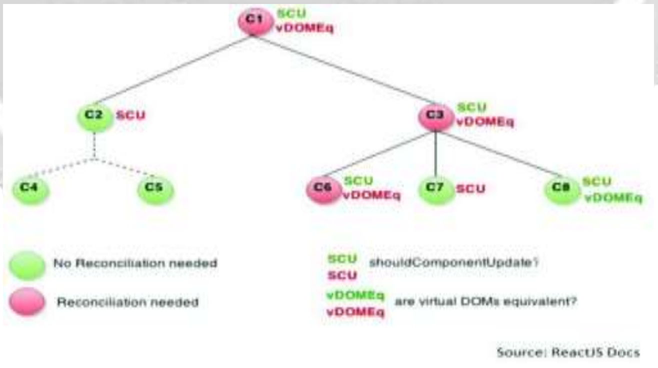

Figure 1: Component Tree for rendering a react component

Think about the progression in Fig 1. For some adjustments of component C1, component C1 chose to re-render. Presently React checks the children in the subtrees of component C1 recursively until every one of the parts in the subtree has been refreshed dependent on the worth produced by the SCU() strategy.

The components in the tree are not compelled to re-render by the parent segment. It implies that the children in that subtree will be evaluated. From the figure, it is outstanding that, since the SCU of C1 is valid, the subtrees C2 and C3 are checked. Since SCU of C3 is valid, its children are assessed, though the parts in the other subtree C2 are not assessed since SCU for C2 is bogus and correspondingly the cycle has proceeded. At long last, the components C6, C3, and C1 are re-delivered.

This type of re-rendering does not cause any issues in the case of a simple application, but in complex applications with many components, this re-rendering causes many performance issues.

And relying only on the React Virtual DOM comparison is not enough. Instead, a few additional measures are required to re-render the components only when required.

The following sections describe a few ways of improving the performance of React applications.

II. LITERATURE SURVEY

The authors[1] have presented the overview of this paper that are the application of objective perspectives on the use of REACTJS that directly and accurately demonstrate the actual usage of making an user friendly UI and a responsive application

III. OPTIMIZING THE REACT APP PERFORMANCE

III. I Search Optimization

Looking through huge informational collections including a huge number of items can a lot of time. In the event that information got from the server isn't all in order, the time intricacy will be O(n). Likewise, creating a single data structure of JSON objects causes an issue of memory over-burdening. Attempting to construct an application with 100,000 array objects may make the application fall flat. The NPM (Node Package Manager) throws a memory mistake. To fix this data can be parted into pieces of fixed sizes and use hashing, with the search attribute as the key.

In this case, the Hash generated using each element is used as the key to point to the Object [2]. Hence when there is a need to search the object based on its name, the Hash for the corresponding string is computed and used to find the object. The complexity of the search will be reduced to O (1).

It tends to be noticed that since we have utilized the hash technique, if data searching using query string "user", then results won't be found. This is because of Hash("user") and Hash("user1") being various values. Thus, the calculation of some more hashes by adding to the pursuit string further is required. For instance, assuming information comprises of the string "user", almost certainly, strings, for example, "user1", "user2", "userx", and so on are available in the informational index. Hence, for the pursuit string "user" developing all the more such strings and looking at the comparing Hashes in the Hash-table is a decent arrangement.

The time complexity of these operations is k * O (1). Where k is the number of combinations of the search string. In comparison to O (n) and even O (n log n) approaches, this provides better performance [3].

If the data set consists of the following strings as Object attributes:

"user", "user1", "user2", . . ., "user22", . . . Assume that the query string is "user".

Also, looking in the Hash-table utilizing the strings each in turn, when a string, say "userx" isn't found in the Hash-table, strings in another structure, say "userxx" can be taken into thought.

In the same way, few other string combinations can be built. The number of combinations built determines the complexity of the algorithm. In sequential search technique, the time complexity will O

(10^n) (assuming the length of the data set to be 10^n) whereas in this case, the time complexity is k * O (1), where k depends on the search string.

This is more similar to an outright search, where data that takes after the search query most precisely is gotten, instead of getting all data with a slight likeness. This is the trade-off that diminishes calculation time [4].

But the main disadvantage is to fix the search attribute. Since the Hash Table is constructed utilizing a specific object attribute, it is beyond to change unless we regenerate the HashTable utilizing an alternative attribute. Even though if the search attribute is fixed, then the approach will be helpful.

III.**II Utilize Existing Component Instances**

For example, let's consider the size of the data set in the order of millions. Creating the instances at once will slow down the application.

To avoid this, a fixed number of instances are created and re-rendered with different Props whenever required.

For example, about 1000 objects can be created initially and whenever the user makes a request, references of the next 1000 objects are passed to the existing component instances [5]. It is recommended to pass the references to the object instances, as performing a copy of the objects will result in duplicating the instances and waste memory.

III.**III Reduce the number of State and Prop variables**

This is an important measure to be taken. By reducing the number of State and Prop variables, the chances of the re-renders of the component that are not necessary can be reduced. Also, frequent and unnecessary changes to the State and the Props can be harmful to the performance of an application. The state must be updated only if it has a visual impact on the application and if not, the state must be updated only at the end [6].

The component could be stopped from delivering until the essential data has been received. This is configured by going over the SCU(props, state) technique for the component. This method chooses if the component should be delivered or not.

Consider a situation where a component receives data by making a **REST API** call to a server and a few local props received from the other components of the application. The component is rendered when it receives the server data. If data from the server is received before the props from other components, this would cause a re- render. Instead, checks in the SCU() method can be added to verify the data, so that this problem can be successfully avoided. It helps to improve the application start-up render time and therefore improve performance[7].

III.**IV Splitting the main component into individual components**

In a code snippet, the component delivers a table where clicking on any row or column would set that component to the state. This implies that each time the cell is clicked, the whole table re-renders which causes performance problems. This is unimportant if the size of the data utilization is little. Yet, in real-time applications, the size of data can be huge, and rendering the whole table on each click is very inefficient [8].

In a code snippet, the component delivers a table where clicking on any row or column would set that component to the state. This implies that each time it's clicked, the whole table re-renders which causes performance problems. This is unimportant if the size of the data utilization is little. Yet, in real-time applications, the size of data can be huge, and rendering the whole table on each click is very inefficient.

This can summarize be summarized as:

- Main Component

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

- Sub-Component

This is an independent component, that is made to handle all the occurring events, without having to interact with other components. Here, each row or column component will have its state, and every time there is a change, react virtual DOM will only compare the state for one particular row or column and re-render it if required. This is an improvement over the previous case, where an entire table was re-rendered. Since React provides two-way data binding, an additional event is not required in the child element to publish the result back to the parent.

III.**V Multithreading**

In most cases, web browsers spawn one thread per tab opened and this thread is responsible for all the operations performed in the application. Hence, if there are many computations to be performed, the thread would have to stop all other operations and this leads to unresponsiveness in the application during this time [9].

But recently, Google and Mozilla have introduced Web workers to make browsers more powerful. A web worker is a JavaScript program that runs on a separate thread in parallel to the main thread, therefore, enabling us to implement Multithreading in the application and use the parallel method of execution [10].

Examples of operations that could be performed are:

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

Consider a situation where there is a requirement for filtering the data in the application. If the size of data utilized by the application is small, multiple need not be running. Yet, assume that the size of data is very large, it isn't effective for a single string to channel the whole information. In these cases, various web workers are made and allocated a particular segment of the whole information to each web workers. Every workers would now channel the segment of data appointed to it and store it in a particular space. At the point when all the web workers complete the assigned work, the main application thread just merges the pieces of the filtered data and obtains a result [11].

In the same way, web workers can be used to perform other background time-consuming tasks, while the application functions well without any performance degradation.

It is important to note that any time a web worker is created, it takes control of some system resources, and these web workers last until the main web worker dies. So, it is important to make sure that we do not overkill by creating many web workers. Browsers such as Mozilla Firefox support up to 512 web workers running simultaneously.

IV. CONCLUSIONS

React is a very useful framework having its own way of tackling performance issues that Web Applications commonly face. But in standalone applications, the performance can still be degraded when complex applications are required to be designed where the application deals with a lot of data processing [12]. This may lead to application response issues and lag. This is a very common problem in large-scale Enterprise Applications. This paper proposes some ways of tackling such problems within the web application.

The techniques described are pointed toward taking out repetitive calculations and help in improving the presentation of the application. While a few methods disclosed are explicit to React structure, different procedures, for example, inquiry improvement and multithreading can be executed in any framework [13].

REFERENCES

1. S. Fröstl,Sebastian."AngularJS performance tuning for long lists."tech.small-improvements.com (2013).

2. C. Wohlie et al., "Experimenting in Software Engineering" in, Springer, 2012. (pp. 1-10).

3. ReactJS Docs reactjs.org/docs, “Optimizing Performance.” (2016).

4. Noam Elboim, blog.medium.com, "How to greatly improve your React App performance." (2018).

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

6. Darrel Greenhill, Jack Francik, Jay Kiruthika, Souheil Khaddaj. “UX Design in Web applications: An approach to improve reuse in web applications”, In Web Engineering (pp. 335-352). Springer, Berlin, Heidelberg, 2016.

7. C. Ebert, M. Kuhlmann and R. Prikladnicki, "Global Software Engineering: An Industry Perspective," in IEEE Software, vol. 33, no. 1, pp. 105-108, Jan.-Feb. 2016.

8. Network, Mozilla Developer. "Web Workers API." (2015).

9. A. Javeed, "Performance Optimization Techniques for ReactJS," (2019) IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 2019, pp.1-5