Neeta Diwan

Optimising performance in React apps

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and communication technology

Bachelor's Thesis

6 May 2024

Abstract

Author:	Neeta Diwan
Title:	Optimising performance in React apps
Number of Pages:	29 pages
Date:	6 May 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and communication technology
Professional Major:	Software Development
Supervisors:	Janne Salonen (Director of School)

App performance is a critical aspect of app development. For an app to be performant, it must provide uninterrupted user-interactions and smooth UI transitions. Although ReactJS is capable of building efficient user interfaces, it takes implementing ReactJS the right way to avoid performance issues.

The purpose of this study is to present app performance optimizing techniques to manage component renders and commonly encountered issues caused by rerenders as well as the amount of render time. In order to mitigate app performance problems, techniques such as memoization and code splitting are known to be efficient and effective. The intent of the study is to make use of these approaches and thus experiment with lazy loading and performance hooks, useMemo and useCallback, provided by ReactJS to optimize app performance.

The study provides insights into identifying and minimizing performance issues by using the techniques mentioned above. A description of React rendering mechanism is also introduced in this thesis. React developer tool Profiler is used to evaluate app performance.

The study results show that optimising React app with useMemo, useCallback and lazy loading techniques improves render duration and loading time. Considering that the given app is data heavy with computations that involves working with large lists, the results from this study could be used as a reference to estimate the responsiveness of data-consuming React apps. The study identifies that the effectiveness of these techniques and performance of React app also depends on how the app is structured and the amount of data and computationally expensive operations it is performing.

Keywords: React, hooks, useMemo, useCallback, lazy loading

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

List of Abbreviations

1 Introduction 1

2 React 2

2.1 Components 2
2.1.1 Composition of components 3
2.1.2 UI elements 4
2.1.3 Components and UI 5

3 Data flow 6

3.1 State 7
3.1.1 Hooks 7
3.1.2 Props 8

4 Rendering mechanism 8

4.1 Virtual DOM 8
4.2 Rendering and Updating UI 9

5 Performance considerations 11

6 Performance issues 13

7 Performance optimising techniques 13

7.1 Memoization 14
7.2 Code splitting 15

8 Tools to evaluate performance 17

8.1 Profiler 17

9 Project overview 17

9.1 Determining correct optimization technique 21
9.1.1 Optimizing performance using useMemo hook 22
9.1.2 Optimizing performance using Callback hook 23
9.1.3 Optimizing performance using lazy loading 24

10 Results 25

11 Conclusion 25

References 27

List of Abbreviations

API: Application programming interface. It is a way for two or more computer programs or components to communicate with each other.

App: Application. Application refers to a software program that's designed to perform a specific function directly for the user or, in some cases, for another software program.

DOM: Document Object Model. It is the data representation of the objects that comprise the structure and content of a document on the web.

JSX: JavaScript Syntax extension. It is used to write HTML in React.

React: React.js. It is a JavaScript library for building user interfaces and is also known as React.js.

1 Introduction

The importance of building apps that deliver high performance cannot be understated. The development of various frameworks and libraries is the result of having recognized the need for a better solution or a product with specific implementations. Improved performance intended to solve certain problems to meet specific objectives is what these technologies are designed to offer.

React gained popularity for its efficient mechanism of building dynamic user-interfaces composed of components. It was deployed on Facebook in 2011 and on Instagram in 2012. React is maintained by Meta engineering team and was created by a Jordan Walke, a software engineer working for Meta, formerly called Facebook [1].

The purpose of this study is to evaluate React techniques to improve performance of React frontend apps. The study focuses on optimizing apps based on functional components and particularly evaluates performance related to User-Interface specific interactions.

This study also discusses the underlying mechanism of React to build front-end web applications. Its capability to create efficient and dynamic web apps has convinced developers to build UI apps using React. As a result, its usage, the number of React developers and React-based websites have been constantly growing.

However, in order to build performant apps based entirely on React and utilize React to its best potential, good knowledge about the performance-enhancing features available in React is undeniably a requirement. At the same time, it is advisable to have an understanding about the mechanism on which React operates. Although React API is at par in terms of performance, when it comes to building data-intensive or complex apps, writing inefficient code can interfere UI performance. This study will delve into some useful techniques provided by

React to optimize React apps and would attempt to identify performance issues and specify solutions.

The following sections provide an insight into how React works and what issues could occur on UI and why. Hence the obvious, what to prevent?

Based on the theory and research, this study would further evaluate performance related to UI interactions by testing an app performance using the React developer tools, Profiler tool. The findings would provide useful insights for React developers and a conclusion to this study.

2 React

React is a JavaScript library used to create user interfaces by combining reusable components [2]. React is based on the single-page app approach. As such, “the presentation layer for the entire application has been factored out of the server and is managed from within the browser” [3]. It is a web development approach that makes an entire app run on a single page. Once the application is loaded it does not require full-page refreshes and content is updated by means of component swapping. This way building dynamic UI web apps is a preferred choice as it enables efficient user interactions and faster UI updates.

2.1 Components

Components are considered as UI building blocks of React. React components can be described as “a piece of the UI (user interface) that has its own logic and appearance” [4]. for example, a thumbnail, a like button, or a video etc. In essence, React components work like JavaScript functions that takes props and return React elements using JSX. These React elements are used to describe what to render on the screen [5]. In other words, React app screens are composed of several components arranged together to form a view or a page. The emphasis is on the composite nature of React components paradigm that is

fundamental to React app development. The aspect of React components is further discussed in the subsequent sections.

2.1.1 Composition of components

Considering the construct of React app in terms of design patterns, it can be said that React's component-based mechanism reflects similarities with composition pattern.

Design patterns are widely used in software development as they are based on solid design practices. Its' fundamental purpose is to write efficient code to accomplish specific goal. Structural patterns, are the set of design patterns that are used to "compose groups of objects into larger structures, such as complex user interfaces or accounting data." Composition design pattern is one of the important Structural patterns [6]. Composition pattern is defined as "the process of solving a complex problem by breaking it down into smaller components (or parts) and then assembling them." [7]. According to Object-oriented development principles, "Composition means that an object is built from other objects" and represents a "has-a relationship" [8]. Figure 1 reflects this trait of composition in React components, where listItem 'has' a image and a 'button', information section further can contain different sections. The object-oriented paradigm looks at it as a mechanism for object reuse [9].

The effectiveness of such composition depends on specific situations, similarly, the effectiveness of React components also depends on the way components are implemented in an application. Figure 1 illustrates components composition in app UI.

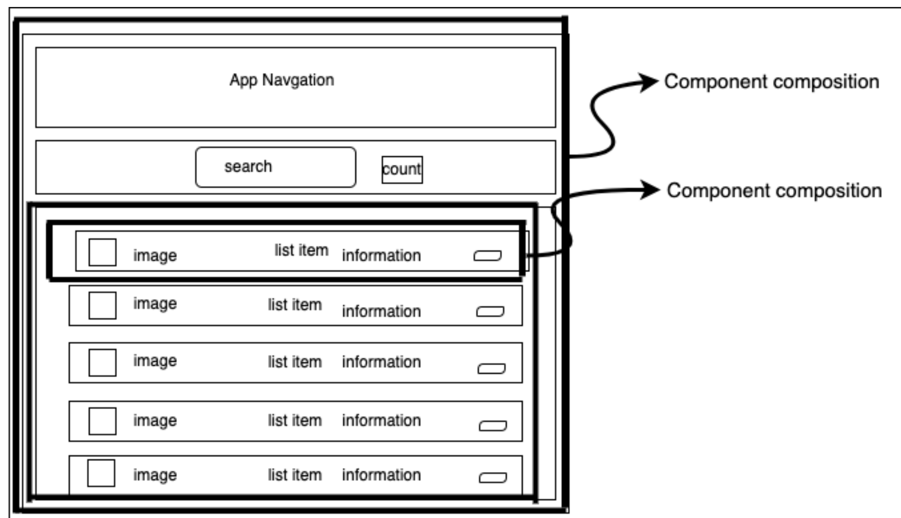

Figure 1 App UI composed of components

Although React gets appreciation for the way it is engineered to deliver fast and seamless UI interactions, it encounters certain limitations as the app grows. Apart from implementing appropriate performance optimizing techniques, that will be examined as the study proceeds, designing app structure in a manner that reduces complexities and provides efficiency is also important. Over complicated and unnecessary component implementations tend to affect app performance as well as makes it difficult to maintain and scale app. The key is to apply composition with appropriate level of granularity so that it achieves a balance, simplifies maintainability, and improves performance [10].

2.1.2 UI elements

UI refers to user interface and can be defined as a point of interaction between humans and computers [11]. A web application's responsiveness and visual elements such as screen, forms, pages, buttons, menu items are all part of UI [12]. These UI elements adds interactivity to the user interface. Figure 2 lists examples and categories of UI elements.

Navigational elements	Input controls	Informational components	containers
used to navigate an interface	enable users to input information	used to communicate information to the user	organise content into sections
- slidebars - search fileds - back arrows	- buttons -checkboxes - text fields	- progress bar	- accordian menu to hide and show content

Figure 2 UI elements categories

In React each element is a plain JavaScript object that describes the component it represents, along with any relevant props or attributes.

2.1.3 Components and UI

While building composite components is a convention, designing React app's overall structure efficiently requires thinking in terms of components. Whereby, UI elements can be seen in terms of components to determine what parts of the UI can be composed, reused and reorganized as components. This approach can help in properly organizing app structure and provides a high-level view of data flow. A visual representation of the UI can serve as a blueprint to identify components and construct component hierarchy. A UI contains elements that are distinct as well as elements that can be reused multiple times in different parts of the interface [13].

React app development being a component-based approach encourages this practice of creating components based on Separation of concerns or single responsibility principle. According to this principle, "The goal is to more effectively understand, design, and manage complex interdependent systems, so that functions can be reused, optimized independently of other functions, and insulated from the potential failure of other functions" [14]. Components can be referred as containers that hold state and props, which are used to store and share data across components.

The discussion in the following section further examines the process involved in State and Dataflow in developing interactive web apps.

3 Data flow

In order to have a clear understanding about the way React app functions, it is important to understand how data is shared, managed and updated. An explanation about how data is shared across components can be approached by learning the rules about data flow in React.

In React data flows in a unidirectional way and is referred to as 'one-way data flow'. This implies that data flows from parent components to child components and not vice versa. To be more specific, it works like, "Data flows down (or downstream), and events flow up (or upstream)" [15]. Figure3 illustrates the concept of unidirectional data flow as described.

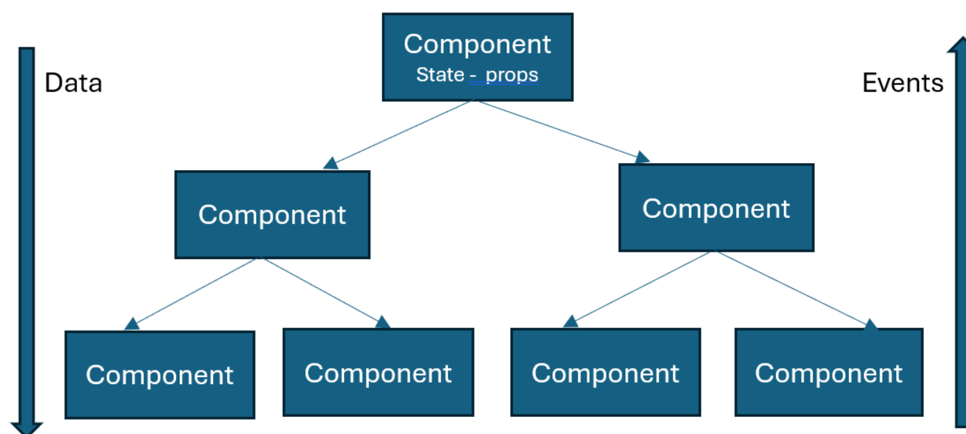

Figure 3 Unidirectional data flow

React features that are fundamental to the process of managing data are State and Props and are described further as subtopics in this data flow section.

3.1 State

State can be defined as “an object that holds information that may change over the component’s lifetime. It is the current snapshot of data stored in a component’s props” [16]. In this sense, State can be referred as component’s memory. In order to reflect a change in UI, a change in component state variable is required. State is private to component and is used to retain data between renders. Stateless variables do not update value to be displayed on UI [17]. This data is passed among components by the means of props. The subsequent sections provide overview of React state management features, involving React hooks and props.

3.1.1 Hooks

Hooks are the JavaScript functions that allow managing state in React apps. Hooks are defined as “simple JavaScript functions that allow components to use the local state and execute side effects (or cross-cutting concerns) and other React features” [18]. Basically, hooks remove the need to use classes and allows using stateful logic between components. Hooks such as, useState and useReducer, are used to manage component state locally and useContext hook is used to manage application-wide data. As for specific usecase, useState is used for simple transformations, while useReducer is used for complex state logic [19].

This implies that a state can be initialized and updated using the useState hook. This study will use useState hook to handle state change in components used for example app. useState hook returns an array consisting of a state variable that retains the data between renders and a setter function that updates the variable. This is what leads to re-rendering a component.

More information about hooks relevant to this study will be provided while discussing performance techniques.

3.1.2 Props

Props are used to share data between parent components and child components. It is important to note that Props are immutable. Props are specified in component instance and a component can contain any number of props with any type of JavaScript data or an expression that evaluates to a value or function [20]. As already mentioned, props can also hold a current snapshot of data, thus props are responsible for passing the updated state variable values to the component tree as required.

4 Rendering mechanism

4.1 Virtual DOM

“The virtual DOM, like the DOM, is an HTML document modelled as a JavaScript object” [21]. Although React DOM is similar to actual browser DOM, it is faster, lightweight and is a virtual representation of the actual DOM in memory i.e. it exists in memory only, while the actual DOM is the real concrete structure of a web page [22]. Virtual DOM functions as an intermediary layer where all changes in the state are evaluated before it syncs with the browser DOM for changes to appear on the UI.

Virtual DOM allows to create user interfaces in a more efficient and performant way, solving problems associated with actual DOM. Updating actual DOM tends to be slower and expensive, due to the fact that every time a change is made to the real DOM, the browser has to perform several operations that can be resource-intensive and time-consuming [23]. The efficiency of virtual DOM is attributed to the fact that it prevents recreation of entire DOM tree and instead only allows to make specific updates to the actual DOM as per the changes introduced in JSX elements [24]. React accomplishes this by using Diffing algorithm that keeps track of the old and new versions of the updates made to virtual DOM. It is the Diff algorithm that holds the responsibility of determining the necessary changes required to update the real DOM and apply those changes

efficiently by batching operations, i.e. one update for one batch instead of multiple updates to the actual DOM, for each state change detected [25].

The specificities of virtual DOM as mentioned above, indicates that virtual DOM allows UI to stay in sync with the changes in state or any UI generated changes, thereby resulting in updating the UI faster. This certainly is one of the features that gives ReactJS apps an edge.

4.2 Rendering and Updating UI

One of the significant aspects of React app development that must be understood is the process involved in updating UI. The official React documentation classifies it into three phases representing user-initiated events, DOM evaluation and UI updates, as illustrated in the figure 4.

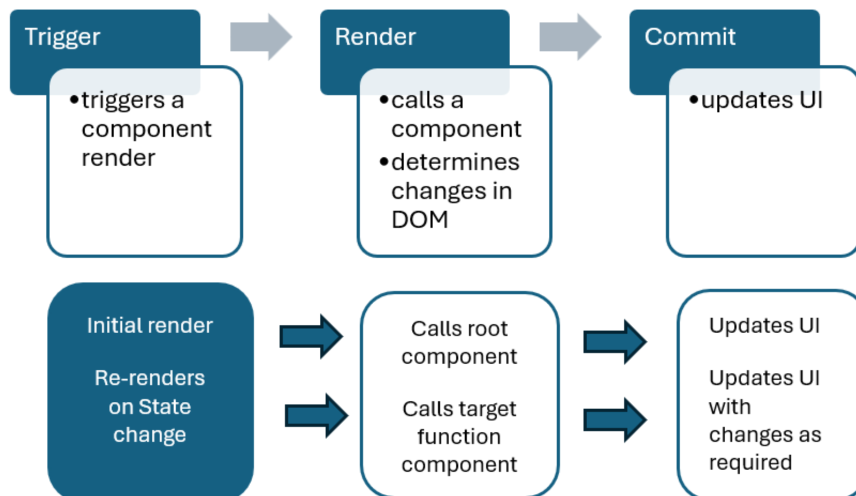

Figure 4 Initial render and rerenders according to UI updating phases

The trigger phase refers to triggering a component to render. This happens on initial render or as a result of state change in a component or its parent component. So, whenever a state is updated with state setter function it triggers component re-render.

This is followed by rendering phase. Component rendering refers to a situation when a component gets called after a code contained in a component or a parent component gets executed. This is an intermediary stage, that determines if there is a need to modify DOM by evaluating previous and current version of Virtual DOM. Rendering phase thus, represents calling components and evaluating what to display on the screen. On the initial render, React calls the root component and create DOM nodes for all React elements returned from component's JSX. Thereafter, for subsequent renders, React calls the function component whose state update triggered the render. During a re-render React performs a check and calculates if there is any difference in the DOM element properties by comparing it with the previous render. It is important to note that the component re-renders involve calling all the nested components and the nested components contained within nested components [26].

The following figure 5 shows example code for how React initially renders a React component or React app. This belongs to index.js file where the code to initialise react app is placed.

```
JS App.js  M    JS index.js M ×
notes-app > src > JS index.js > ...
1   import React from 'react';
2   import ReactDOM from 'react-dom/client';
3   import App from './App';
4
5   const root = ReactDOM.createRoot(document.getElementById('root'));
6   root.render(
7     <React.StrictMode>
8       <App />
9     </React.StrictMode>
10  );
```

Figure 5 Initial render for a fully built app in React

Referring to the explanation provided in React official documentation, React app is initially rendered by “calling ‘createRoot’ with the target DOM node”, and then

”calling its ‘render’ method with the app component” [27] ‘createRoot’ allows to create a root element to display React app components inside a browser DOM node. A call to ‘createRoot’ is chained with a call to ‘render’, i.e. `root.render´ that takes a React component, typically <App/> to display its React content. A fully React based app typically has one createRoot call for its root components [28].

Proceeding with a description about each phase, it might be useful to reiterate that rendering phase deals with communicating with the Virtual DOM. Its’ task is to confirm if there is any necessary change to be committed to the real DOM, which if applicable updates the DOM and UI accordingly in the commit phase. This process updating the DOM is referred as ‘committing’.

The initial render in the commit phase has React implement appendChild() DOM API to update the screen with all the existing DOM nodes. As for re-renders, React updates the DOM with latest version of rendering output as calculated in rendering phase. Most notable aspect of the process of updating UI is undoubtedly that “React only changes the DOM nodes if there’s a difference between renders”, a statement from React official documentation.

5 Performance considerations

Web performance involves building websites and apps that are quick to load and responsive to user interaction [29]. Web performance is not only about metrics, but it also “includes both objective measurements like time to load, frames per second, and time to become interactive, and subjective experiences of how long it felt like it took the content to load“ [30]. Recommendations for building performant apps include minimising loading and response times and using techniques to conceal latency by making the user experience as available, interactive and quick as possible [31]. Figure 6 lists main aspects related to web performance.

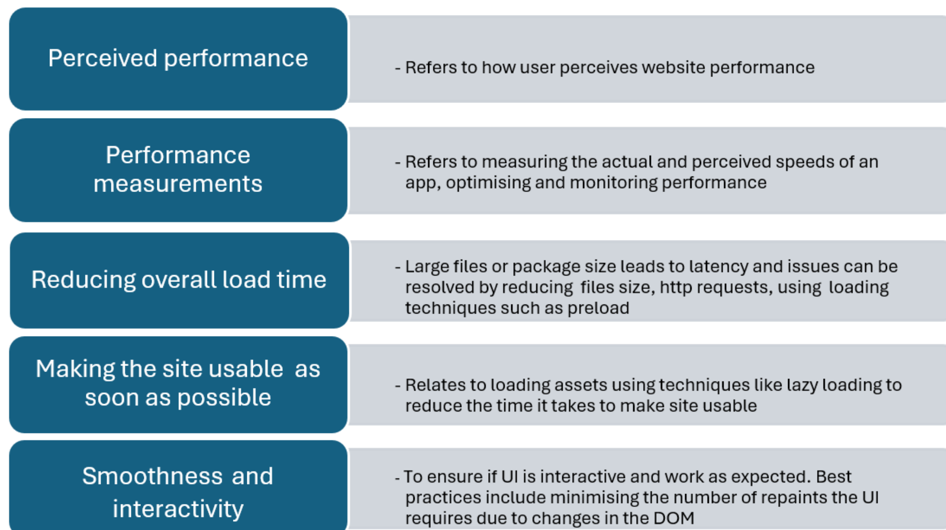

Figure 6 Web app performance considerations

This conveys that to make web apps performant it is important to ensure that apps are quick to respond without any delays in content availability. In order to evaluate app performance there are certain suggestions pertaining to acceptable response time for web and apps. The research reveals that the page loading within 1 second is considered as recommended app performance level for a good user experience.

💡 If a site takes >1 second to become interactive, users lose attention, and their perception of completing the page task is broken [Source: Google Developers Blog]

Figure 7 Performance indicator

Web app performance can be measured using several metrics. It is suggested to measure performance that reflect exact figures instead of general perception based approach [32]. This study would analyse app performance using React developer tools Profiler that would help in clearly distinguishing between app performance before and after optimization.

6 Performance issues

As discussed earlier, React app runs on single page with a UI composed of several components. React by default is based on mechanism that allows to build fast interactive frontend webapps, however the issues tend to surface when the app gets larger. The large amount of JavaScript bundle that gets downloaded and render can cause performance problems. The app could become slower already on the initial render and would additionally make UI interactions inconsistent on subsequent renders [33].

In terms of renders and rerenders, rendering components that handle expensive computations are most time consuming and affects apps' responsiveness. The time for app to become responsive tends to increase as the number of expensive components to be rerendered increases.

To sum up, it can be deduced that renders, rerenders and bundle size have a direct impact on app performance in terms of the time it takes for an app to become interactive and responsive.

7 Performance optimising techniques

According to Official React documentation, "A common way to optimize rerendering performance is to skip unnecessary work. For example, you can tell React to reuse a cached calculation or to skip a re-render if the data has not changed since the previous render" [34]. This statement highlights reusing cached results thereby avoiding component rerenders if the same input is used before. Memoization is a technique that works on this concept of reusing and caching the same inputs for computation intensive function calls [35]. Based on this approach React provides hooks, named as useMemo hook and useCallback hook, specifically to facilitate optimizing app performance. React also allows the creation of custom hooks to build implementations as per project requirements. This study examines the difference that memoization and code splitting

techniques make to app performance by addressing the issues that affect performance.

7.1 Memoization

Memoization is described as an optimization technique used for “speeding up web applications by caching the results of expensive function calls. It returns the cached result when the same input arguments have been passed again” [36]. In other words, it improves app performance by caching the results of expensive function calls. The main advantage is that it does not cause unnecessary re-renders if the cached result is the same for the latest input arguments. However, memoization is recommended for expensive calculations such as sorting and filtering operations during rendering and is deemed redundant for simple calculations within functions [37].

As for the useMemo hook and useCallback hook it is to be noted that although useMemo and useCallback hooks work to provide performance benefits, they have different use cases. While useMemo hook allows caching the resulted value of an expensive calculation, the useCallback hook is meant to cache a function definition [38].

Figure 8 and 9 shows how useMemo and useCallback hooks are implemented in practice respectively.

```
const memoizedValue = useMemo(() => a+b, [a,b]);
```

Figure 8 useMemo hook

```
const memoizedFunction= (() => doSomething(a, b), [a,b]);
```

Figure 9 useCallback hook

It is worth mentioning that using these hooks all over the app or applying memoization wrongly could adversely impact app performance as these techniques use memory and could increase overhead costs. To clarify the difference between and suitability for each hook as per usecase, figure 10 provides an overview of these techniques specifying their role and usage.

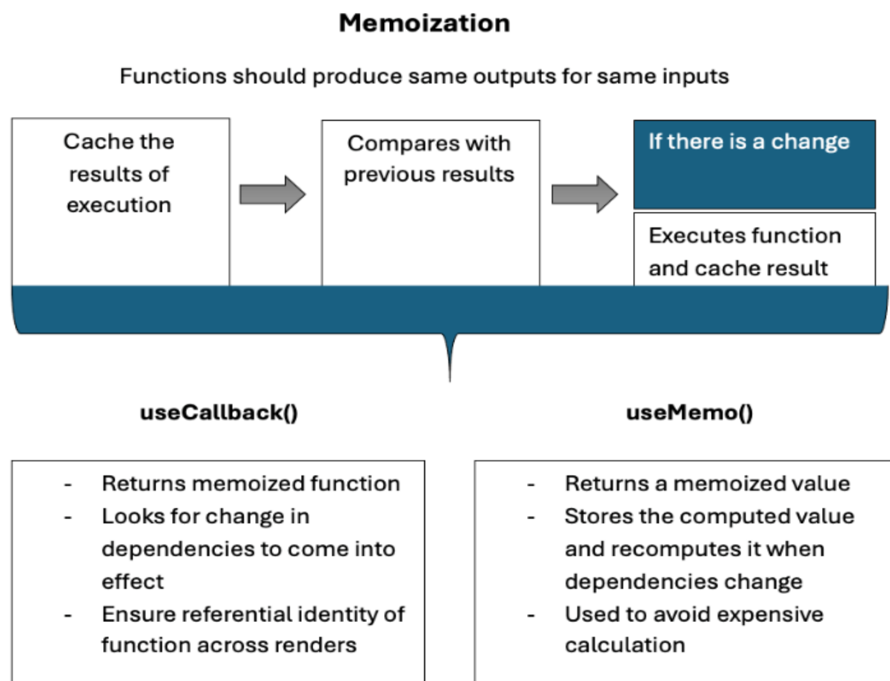

Figure 10 Memoization and difference between useMemo and useCallback hooks

Another important technique that considerably improves app performance is code splitting. The following section explains the concept, its usage, and implementation.

7.2 Code splitting

Referring to the phases involved in updating app UI, this section relates to the Initial phase, render. As mentioned, React loads JavaScript files on the initial loading of a web app, if the bundle is large it can hamper app performance.

Code splitting can be used to solve this issue. Code splitting involves breaking the main code for an app into multiple smaller bundles. A bundle gets loaded only when any of the components it contains is needed. This technique of lazily loading bundles is referred as lazy loading [39]. The main purpose of lazy loading is thus that it ”defers the loading of noncritical JavaScript until after the page has loaded.” [40]. Thus, by splitting app code it is possible to reduce page load times and data usage as it only loads the code needed for a particular page or feature.

React offers implementations such as React.lazy and Suspense to address page loading time issues. It is a standard technique commonly used to optimize large apps with several components to cause a delay to load the component that are not required on the initial render [41]. The following figure shows the syntax for implementing lazy loading.

```
const lazyLoadingComponent = React.lazy(() => './lazyLoadingComponent');
```

Figure 11 Lazy loading component

Suspense is used in conjunction with lazy loading to display content such as, loading indicator while the requested content is being loaded. It uses fallback prop for this purpose and accepts any React elements to be rendered before the component gets loaded. By convention, lazy components are rendered inside Suspense component.

```
<Suspense fallback={<div>Loading...</div>}>
{/*    lazy load component */}
</Suspense>
```

Figure 12 Suspense and fallback prop

It is also advised not to opt for any optimization if the performance improvement is not significant enough to have a noticeable impact on user experience [42]. At the same time, the author also encourages having some objective measures in

place to take decisions whether an app requires any performance tuning. It is recommended to measure app performance based on metrics and making performance evaluations according to accurate results.

8 Tools to evaluate performance

Web app performance can be evaluated using several metrics and tools. This study would use React Developer tools to test app performance, identify performance problems and causes and use that information to apply correct optimization technique.

React developer tools provide valuable insights that can help identifying issues that impact app performance. For the purpose and scope of this study, React developer tools Profiler is used to evaluate app performance.

8.1 Profiler

Profiler is a tool available in React developer tools that measures how frequently a react components render and the time components take to render, thereby indicating which components are slow. This tool will be used to gather information about the render duration and performance issues. The results of the optimized version and the unoptimized version of the app will be used to make performance evaluation.

9 Project overview

A demo app consisting of large set of data is built to evaluate performance. The data comes from a dummy data provider API. App performance is analyzed by interacting with UI, particularly implementing search and button specific events.

The app contains two main components, Posts and Archive, that display a list of blog posts. Post component is a parent component and Archive is a child component. Post and Archive both hold PostItem component to display array of

posts list items as a list. Parent Component, Posts, displays large list of posts. The Archive component represents a data-intensive expensive component contained in Parent component of the App. It is to be noted that the component composition used in this study is not applicable for a realistic app. It is for demonstration purpose only to address common performance issues.

As explained in the theory sections, expensive calculations in any scenario have performance implications. Also, a change in state in parent component causes all its child component to re-render and could result in slowing down the app if the component being rerendered is an expensive component.

The following section presents the performance results of implementing useMemo, useCallback and lazy loading to handle slow components with expensive operations and reduce renders and app load time, as discussed in this study. The screenshots of the results provided by Profiler illustrates the amount of time consumed by components to render and the cause for render. The UI actions performed are Search and deletion.

Following are the screenshots of data provided by Profiler representing App performance on initial render without optimization.

Figure 13 shows Posts and Archive components get rendered on initial loading

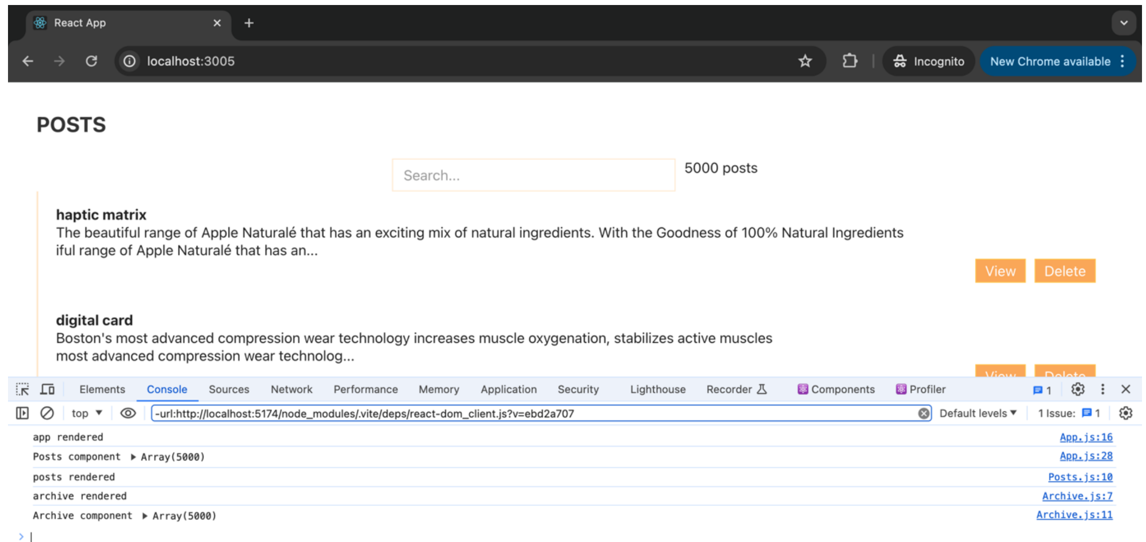

Figure 13 Components render on initial loading

The following screenshot shows the time it took for components to render on initial load, i.e. 804.1ms

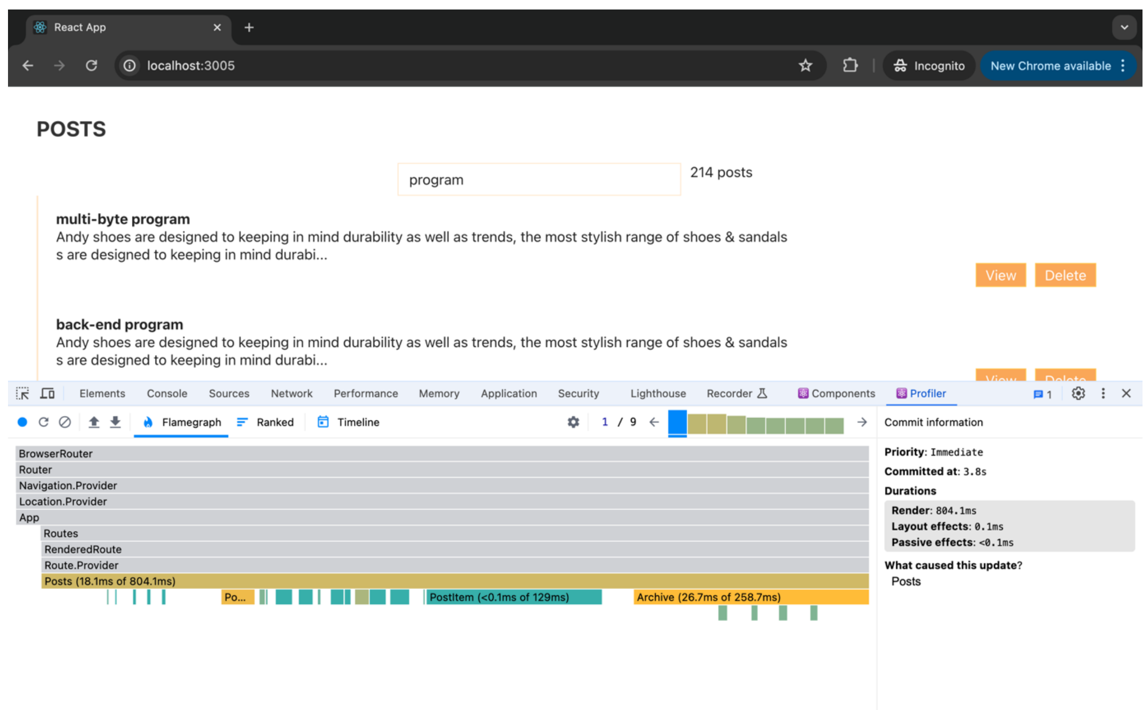

Figure 14 Component render duration on initial loading

The following graph shows the components that are rendered, displaying the component that took the longest time to render at the top. Likewise, listing all components in order according to the time each took to render. The time shown for Archive and PostItem indicates that these components took the maximum amount of time to render.

Figure 15 Graph displaying all components based on render time

It can be observed from the figure16 that the child component Archive rendered because its parent component rendered i.e. The Search and deletion performed in Posts component caused Archive component to re-render.

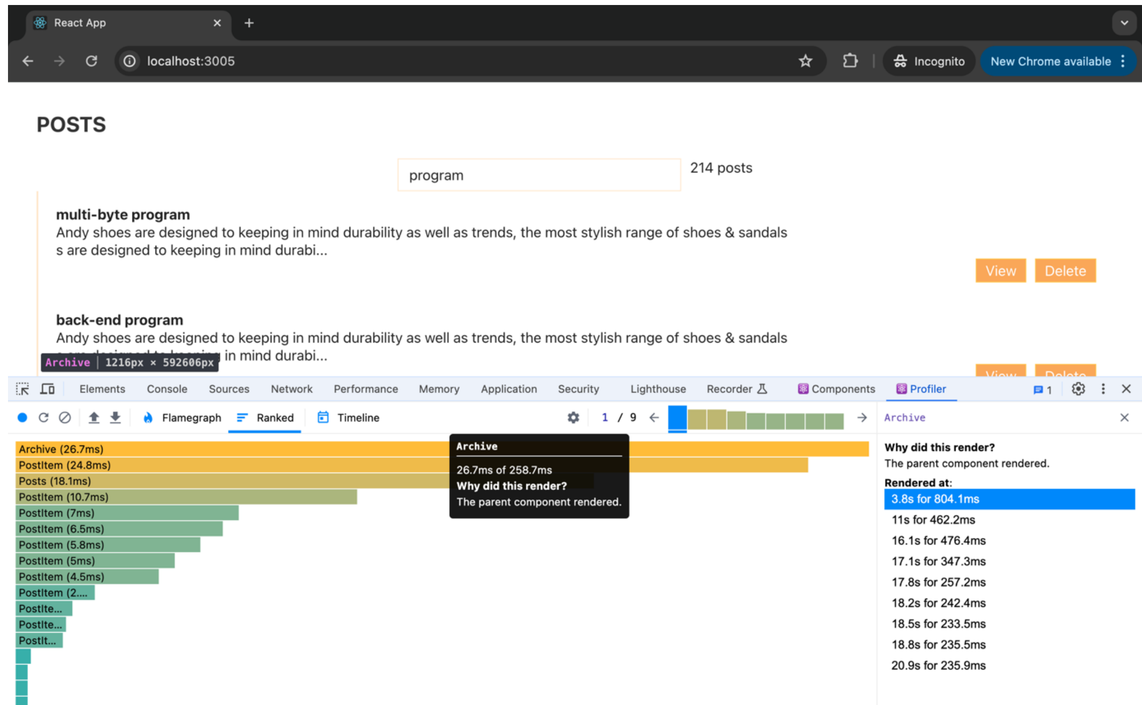

Figure 16 Graph displaying the reason for the component to render

9.1 Determining correct optimization technique

The Profiler performance results indicate problems with the performance of the given app. Referring to the performance consideration as presented in this thesis, the optimal time for a page to load a UI completely should not exceed 1 second. The following section delves into applying optimization to improve app performance by reducing re-renders and app loading time.

As UI interaction involves performing search operation that involves traversing through the large list of posts using filter function. Implementing ‘useMemo()’ will be suitable option as it allows to cache the result, a value returned from functions between rerenders and uses it again if the same search term is used thereby, avoiding re-renders.

Another action performed on the UI is deleting a post. A button click invokes ‘handleDelete()’ function to delete a post from the post collection. As described earlier, the useCallback maintains the referential identity of a function

'useCallBack()' hook, this hook caches a function definition between re-renders. This hook will be used for optimization.

In order to tackle issues concerning app loading time, implementing lazy loading is an effective approach. This will prevent loading the non-urgent content and will only load it when required.

9.1.1 Optimizing performance using useMemo hook

The following Profiler view shows the results of implementing useMemo hook to filter posts as per the search term. The user interface did not appear appropriately responsive to UI interaction although shows improvement in component render time.

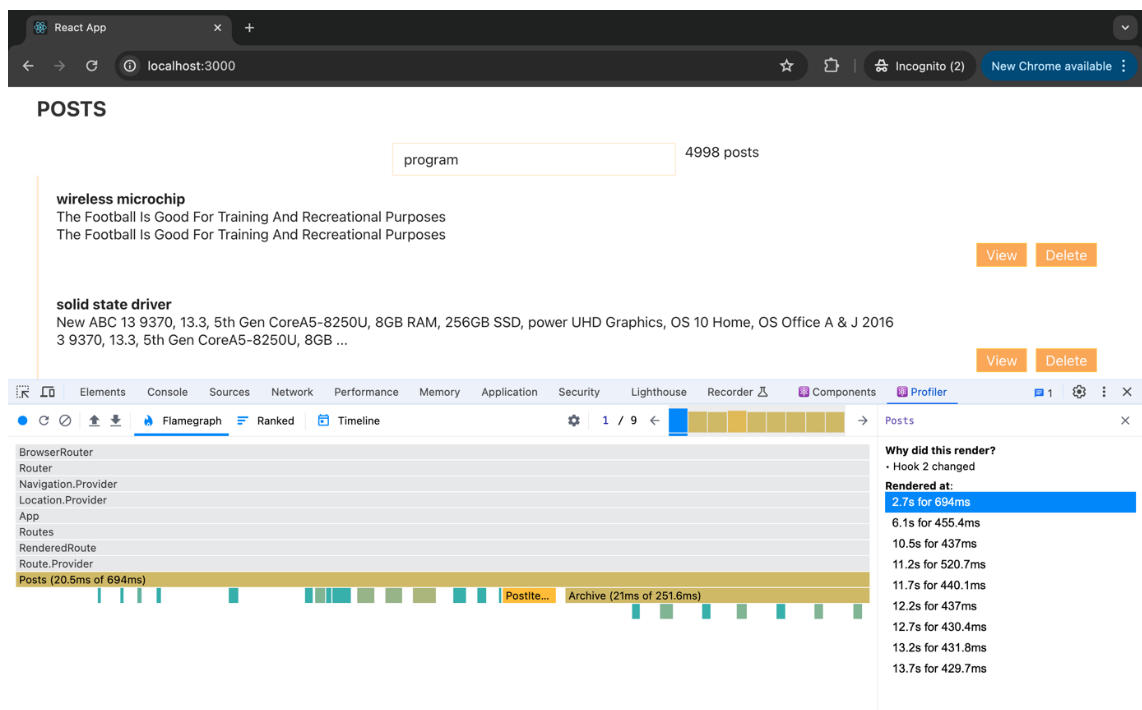

Figure 17 useMemo optimization results

The following figure 18 shows the Ranked view of Profiler indicating the time components took to render. The information clearly reflects improvement in render time.

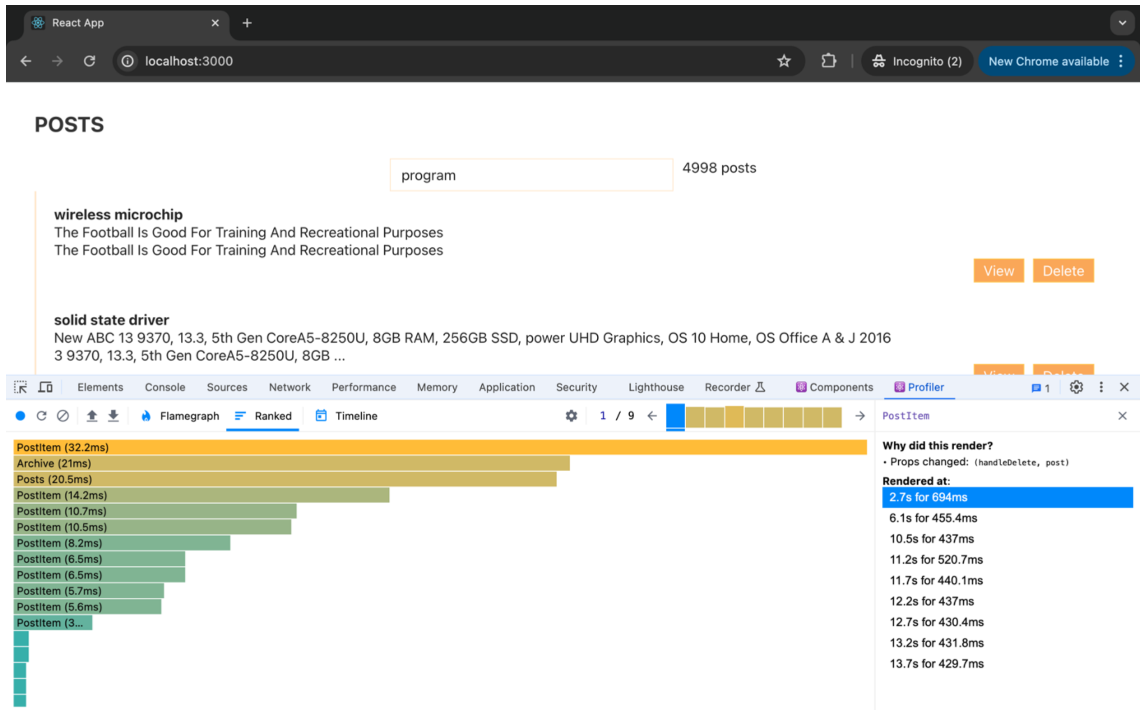

Figure 18 useMemo optimization result

9.1.2 Optimizing performance using Callback hook

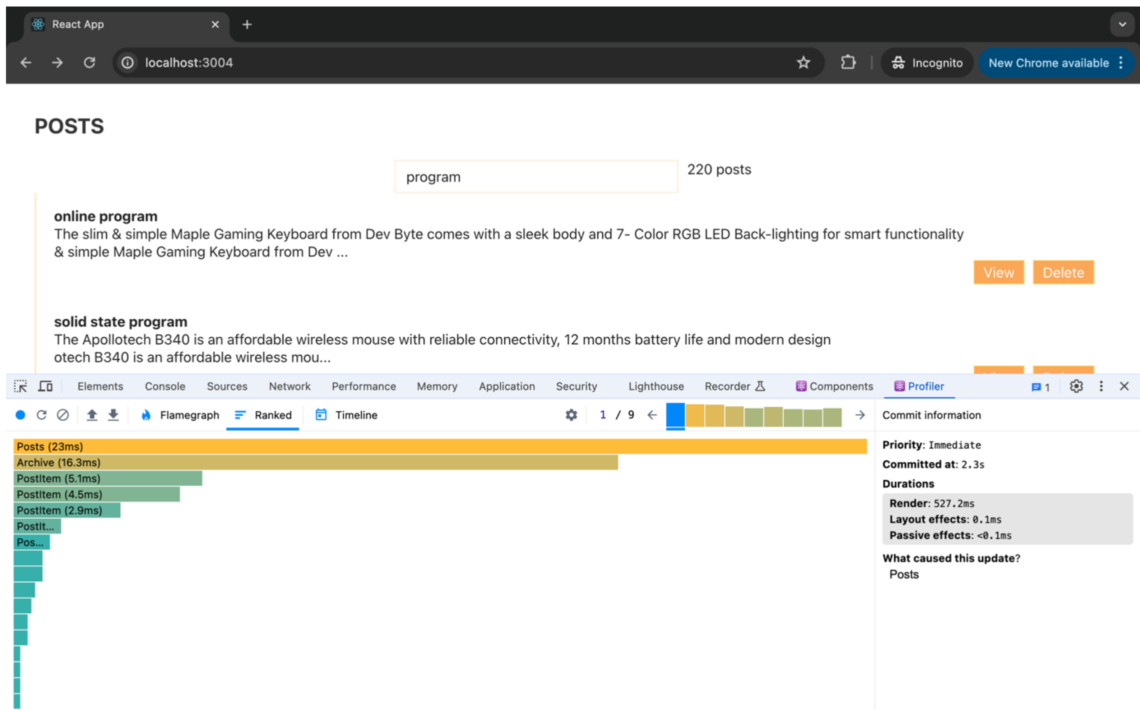

Figure 19 useCallback optimization results

9.1.3 Optimizing performance using lazy loading

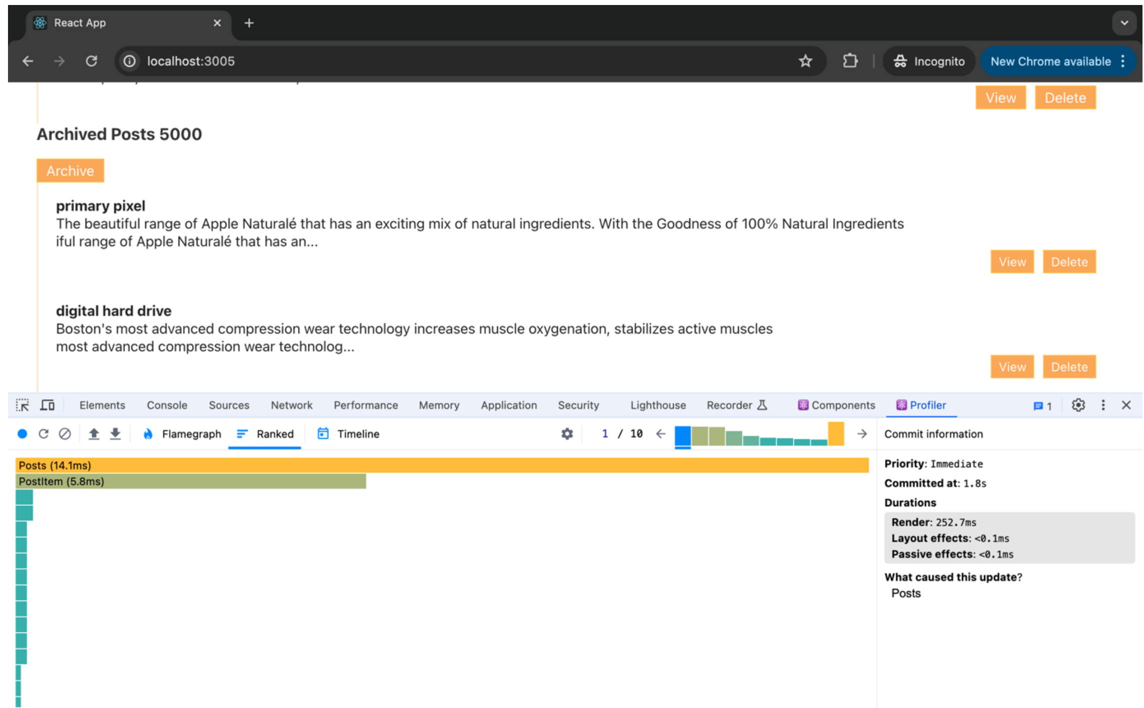

Figure 20 Lazy loading optimization results

10 Results

The following figure 21 shows the comparison of results obtained from Profiler sessions.

	Before optimization	After optimization
lazy loading	804.1ms	252.7s
useMemo hook	804.1ms	694ms
useCallback hook	804.1ms	527ms

Figure 21 Comparison of performance results

By applying lazy loading, intial render time of components on initial loading reduced from 804.1ms to 252.7ms. Similarly, useMemo() and useCallback() show performance improvements as the time it took for components to render also reduced as a result of reduction in number of renders.

11 Conclusion

The results obtained from this study would be sufficient to conclude that React useMemo(), useCallback() and lazy loading techniques provide performance advantage. However, it appears as the app utilizes large set of data and although show a positive performance gain, the app is not performant enough. The objective of the study was to analyse performance improvements using React performance hooks and lazy loading with a focus to achieve efficiency and loading speed by reducing unnecessary renders and rerenders. Reflecting on the

viewpoints and findings presented in this study, it can be said that the core of the solution, i.e. identifying the performance hampering implementations, such as, components that unnecessarily rerender or are slow to render, and determining the right optimizing technique is certainly a prerequisite in the endeavors to build performant apps. Nested components is a way React app structure is organized, excessive use of nested components results in excessive rerenders. Components that perform expensive operations must be evaluated for the possibility for optimization as well as re-renders should be appropriately managed. Nonetheless, it is important to use these techniques only if the outcome is notably positive improvement in app performance.

References

1 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_02.xhtml#_idParaDest-61. Accessed on: 15/4/2024

2 https://react.dev/. Accessed on: 15/4/2024.

3 https://learning.oreilly.com/library/view/spa-design-and/9781617292439/kindle_split_011.html. Accessed on: 25/4/2024.

4 https://react.dev/learn. Accessed on: 11/4/2024

5 https://react.dev/learn. Accessed on: 25/4/2024.

6 https://learning.oreilly.com/library/view/the-object-oriented-thought/9780135182130/ch10.xhtml#ch10lev1sec3. Accessed on: 25/4/2024.

7 https://medium.com/@master_43681/mastering-composition-patterns-in-react-and-typescript-3b503645ab6e. Accessed on: 25/4/2024.

8 https://learning.oreilly.com/library/view/the-object-oriented-thought/9780135182130/ch01.xhtml#ch01lev1sec12. Accessed on: 25/4/2024

9 https://learning.oreilly.com/library/view/the-object-oriented-thought/9780135182130/ch07.xhtml#ch07lev1sec1. Accessed on: 22/4/2024.

10 https://learning.oreilly.com/library/view/the-object-oriented-thought/9780135182130/ch07.xhtml#ch07lev1sec2. Accessed on: 25/4/2024.

11 https://www.coursera.org/articles/ui-design. Accessed on: 13/4/2024.

12 https://www.geeksforgeeks.org/user-interface-ui/. Accessed on: 28/4/2024.

13 https://learning.oreilly.com/library/view/react-in-action/9781617293856/kindle_split_011.html. Accessed on: 29/4/2024

14 https://en.wikipedia.org/wiki/Separation_of_concerns. Accessed on: 6/5/2024.

15 https://learning.oreilly.com/library/view/beginning-reactjs-foundations/9781119685548/c06.xhtml#head-2-96. Accessed on: 2/5/2024.

16 https://learning.oreilly.com/library/view/learning-javascript-design/9781098139865/ch12.html#idm45017690349552. Accessed on: 17/4/2024.

17 https://react.dev/learn/state-a-components-memory. Accessed on: 17/4/2024.

18 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_03.xhtml#_idParaDest-114. Accessed on: 13/4/2024.

19 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_03.xhtml#_idParaDest-115. Accessed on: 4/5/2024.

20 https://learning.oreilly.com/library/view/beginning-reactjs-foundations/9781119685548/c06.xhtml#head-2-97. Accessed on: 28/4/2024.

21 https://learning.oreilly.com/library/view/fluent-react/9781098138707/ch03.html#id33. Accessed on: 18/4/2024.

22 https://www.geeksforgeeks.org/difference-between-virtual-dom-and-real-dom/. Accessed on: 18/4/2024.

23 https://learning.oreilly.com/library/view/fluent-react/9781098138707/ch03.html#id3. Accessed on: 2/5/2024.3

24 https://www.geeksforgeeks.org/difference-between-virtual-dom-and-real-dom/. Accessed on: 2/5/2024.

25 https://learning.oreilly.com/library/view/fluent-react/9781098138707/ch03.html#id33. Accessed on: 2/5/2024

26 https://react.dev/learn/render-and-commit. Accessed on: 2/5/2024.

27 https://react.dev/learn/render-and-commit. Accessed on: 2/5/2024.

28 https://react.dev/reference/react-dom/client/createRoot#root-render. Accessed on: 2/5/2024.

29 https://developer.mozilla.org/en-US/docs/Learn/Performance. Accessed on: 23/4/2024.

30 https://developer.mozilla.org/en-US/docs/Web/Performance. Accessed on: 23/4/2024.

31 https://developer.mozilla.org/en-US/docs/Web/Performance. Accessed on: 1/5/2024.

32 https://learning.oreilly.com/library/view/react-cookbook/9781492085836/ch10.html. Accessed on: 4/5/2024.

33 https://learning.oreilly.com/library/view/react-cookbook/9781492085836/ch10.html#idm45458558019120. Accessed on: 13/4/2024.

34 https://react.dev/reference/react/hooks#performance-hooks. Accessed on: 22/4/2024.

35 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_03.xhtml#_idParaDest-132. Accessed on: 24/4/2024.

36 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_03.xhtml#_idParaDest-132. Accessed on: 2/5/2024.

37 https://learning.oreilly.com/library/view/react-interview-guide/9781803241517/B18603_03.xhtml#_idParaDest-134. Accessed on: 2/5/2024.

38 https://react.dev/reference/react/hooks. Accessed on: 3/5/2024.

39 https://learning.oreilly.com/library/view/react-cookbook/9781492085836/ch10.html. Accessed on: 1/5/2024.

40 https://learning.oreilly.com/library/view/fluent-react/9781098138707/ch05.html#id66. Accessed on: 1/5/2024.

41 https://learning.oreilly.com/library/view/fluent-react/9781098138707/ch05.html#id68. Accessed on: 1/5/2024.

42 https://learning.oreilly.com/library/view/react-cookbook/9781492085836/ch10.html. Accessed on: 2/5/2024.