useDeferredValue Hook

- The useDeferredValue hook in React was introduced to handle situations where you want to delay the rendering of an expensive computation or update while keeping the UI responsive.
- This hook is particularly useful in cases where real-time feedback (e.g., user typing or interactions) needs to be handled smoothly without blocking the browser with computationally heavy tasks.
- The useDeferredValue hook is a powerful tool for ensuring a smooth user experience in applications with complex rendering requirements.
- By deferring less critical updates, it prioritizes responsiveness and interactivity, which are essential for modern web applications.

Key Concepts

- **Purpose:** useDeferredValue helps in deferring a value until the browser has the time to render it, which prevents the UI from becoming unresponsive during intensive operations. It allows prioritizing immediate updates (like user input) over deferred ones.
- **React Feature Integration:** React's Concurrent Rendering enables the splitting of rendering tasks into chunks. The useDeferredValue hook leverages this capability to allow less urgent updates to wait until higher-priority updates (like user typing) are completed.
- **Behavior:** When a value changes, useDeferredValue returns a "deferred" version of the value. This deferred value lags behind the actual value until React has rendered it. It works similarly to useTransition, but instead of deferring rendering directly, it defers the value being rendered.

How useDeferredValue Works Internally : When you pass a value to useDeferredValue, React keeps track of two versions of the value:

- The "immediate" value, which updates right away.
- The "deferred" value, which lags behind the immediate value and updates only when React can schedule it without blocking.

This mechanism ensures React's rendering remains non-blocking, which is a fundamental aspect of Concurrent React.

Syntax

```
const deferredValue = useDeferredValue(value);
```

Example: Search Filtering without useDeferredValue

Here, every keystroke triggers an expensive computation, making the input lag.

App.js

```
import React from 'react';
import SearchComponent from './SearchComponent.js';
function App() {
  const data = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry', 'Fig', 'Grape'];
  return (
    <div className="App">
      <h1>Search App</h1>
        <SearchComponent data={data} />     </div>
  );}
export default App;
```

SearchComponent.js

```
import React, { useState, useMemo } from 'react';
function SearchComponent({ data }) {
  const [query, setQuery] = useState('');
  const filteredData = useMemo(() => { // Use useMemo to optimize filtering
    return data.filter((item) =>
      item.includes(query)
    );  }, [data, query]);
  return (
    <div>
      <input        type="text"        value={query}
       onChange={(e) => setQuery(e.target.value)} placeholder="Search..."/>
      <ul>        {filteredData.map((item) => (
          <li key={item}>{item}</li>
        ))}      </ul>
    </div>
  );}
export default SearchComponent;
```

Problem: The input becomes unresponsive as the dataset grows.

Example: Search Filtering with useDeferredValue

The filtering is deferred, keeping the input responsive.

App.js

```
import React from 'react';
import SearchComponent from './SearchComponent.js';
function App() {
  const data = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry', 'Fig', 'Grape'];
  return (
    <div className="App">
      <h1>Search App</h1>
          <SearchComponent data={data} />     </div>
  );}
export default App;
```

SearchComponent.js

```
import React, { useState, useDeferredValue } from 'react';
function SearchComponent({ data }) {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query); // Defers the query update
  const filteredData = data.filter(item => item.includes(deferredQuery));
// Uses deferred value
  return (
    <>
      <input      type="text"      value={query}      onChange={(e) =>
setQuery(e.target.value)}      placeholder="Search..."      />
      <ul>  {filteredData.map(item => ( <li key={item}>{item}</li>   ))}
      </ul>
    </>
  );
}
export default SearchComponent;
```

note:

- The input (query) updates immediately to maintain responsiveness.
- The expensive filtering (deferredQuery) is deferred until React has time to process it.

Example: Defers the update of the filteredList passed to the ExpensiveList component.

```
import React, { useState, useDeferredValue, useMemo } from "react";
function ExpensiveList({ items }) {
  const deferredItems = useDeferredValue(items); // Defer heavy computation
  // Simulating an expensive computation
  const renderedList = useMemo(() => {
    console.log("Computing expensive list...");
    return deferredItems.map((item, index) => <li key={index}>{item}</li>);
  }, [deferredItems]);
  return (
    <div>
      <p>
        <strong>Note:</strong> The list below updates with a slight delay to keep
the UI responsive.
      </p>
      <ul>{renderedList}</ul>
    </div>
  );
}
function App() {
  const [input, setInput] = useState("");
  const [list, setList] = useState(Array.from({ length: 10000 }, (_, i) => `Item ${i +
1}`));
  // Filter the list based on input
  const filteredList = useMemo(() => {
    console.log("Filtering list...");
    return list.filter((item) =>
      item.toLowerCase().includes(input.toLowerCase())
    );  }, [input, list]);
  return (
    <div>      <h1>useDeferredValue Demo</h1>
      <input        type="text"        value={input}        onChange={(e) =>
setInput(e.target.value)}        placeholder="Search..."      />
      <ExpensiveList items={filteredList} />
    </div>
  );}
export default App;
```

Why useDeferredValue was Introduced

- **Handling Lag during Intensive Rendering:** When a component renders complex calculations or updates based on user input, the UI can freeze or lag. useDeferredValue was introduced to address this by enabling smooth user interactions while deferring less critical updates.
- **Better User Experience:** For example, in a search bar with a large dataset, updating search results on every keystroke might cause delays. useDeferredValue defers the search results' update to ensure the typing remains responsive.

Use Cases

- **Search Autocomplete:** In a search bar that filters a large list, useDeferredValue ensures the input field stays responsive while deferring the filtering computation.
- **Expensive Rendering:** When rendering large components like data visualizations or lists, useDeferredValue can defer the rendering of those components to prevent UI blocking.
- **Smooth Interactions:** In interactive UIs with animations or real-time user feedback, useDeferredValue avoids janky experiences caused by heavy computations.

Key Advantages

- Improved Responsiveness: Prevents UI blocking during expensive computations.
- User Experience Enhancement: Ensures real-time user feedback (like typing or dragging) without delay.
- Simpler alternative to useTransition: Easier to use for scenarios where only a single value needs to be deferred.

useDeferredValue vs. useTransition in React

useTransition: useTransition defers rendering entire components and is ideal for state updates where rendering itself can be postponed.

useDeferredValue, on the other hand, defers values, not components. It's a finer-grained approach, allowing you to delay just the value propagation to child components without changing the parent state immediately.

React introduced these two hooks to improve performance and manage UI responsiveness in scenarios involving heavy computations or concurrent rendering. While both aim to prioritize user interactions over non-critical updates, their use cases and implementations differ significantly.

useDeferredValue

- Purpose: To defer updating a value until higher-priority tasks, such as rendering user input, are completed.
- Usage: Best for deferring computations or updates in child components that depend on rapidly changing state.
- Key Concept: The deferred value updates slightly after the immediate value to prevent blocking the rendering of the UI.

Example: useDeferredValue
This example demonstrates deferring the rendering of a large filtered list when the user types in an input field.

```
import React, { useState, useDeferredValue, useMemo } from "react";

function ExpensiveList({ items }) {
  const deferredItems = useDeferredValue(items); // Defer the heavy computation

  // Simulate an expensive operation
  const renderedList = useMemo(() => {
    console.log("Rendering expensive list...");
    return deferredItems.map((item, index) => <li key={index}>{item}</li>);
  }, [deferredItems]);
  return <ul>{renderedList}</ul>;
}

function App() {
  const [input, setInput] = useState("");
  const [list] = useState(Array.from({ length: 10000 }, (_, i) => `Item ${i + 1}`));

  const filteredList = useMemo(() => {
    console.log("Filtering list...");
    return list.filter((item) =>
      item.toLowerCase().includes(input.toLowerCase())
    );
  }, [input, list]);

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Search..."
      />
      <ExpensiveList items={filteredList} />
    </div>
  );
}

export default App;
```

How useDeferredValue works here

- The input updates immediately as the user types.
- The filteredList computation is deferred slightly in the ExpensiveList component using useDeferredValue.
- The list rendering occurs slightly later than the input field's update, ensuring smooth typing even with a large dataset.

useTransition

- Purpose: To create "transitions," allowing React to treat state updates as low-priority and letting more urgent updates (like input field rendering) happen first.
- Usage: Best for triggering low-priority state updates that can wait, such as navigation between pages or rendering filtered data.
- Key Concept: Separates urgent state (e.g., input value) from non-urgent state (e.g., filtered data) using a startTransition function.

Example: useTransition

This example shows how to prioritize user input over expensive list rendering during a transition.

```
import React, { useState, useTransition } from "react";

function App() {
  const [input, setInput] = useState("");
  const [list, setList] = useState(Array.from({ length: 10000 }, (_, i) => `Item ${i + 1}`));
  const [isPending, startTransition] = useTransition();

  const handleInputChange = (e) => {
    const value = e.target.value;
    setInput(value);

    // Start a low-priority transition
    startTransition(() => {
      const filtered = list.filter((item) =>
        item.toLowerCase().includes(value.toLowerCase())
      );
      setList(filtered);
    });
  };

  return (
    <div>
      <input type="text" value={input} onChange={handleInputChange}
        placeholder="Search..." />
      {isPending && <p>Updating list...</p>}
      <ul>
        {list.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
export default App;
```

How useTransition works here

- Input Responsiveness: The input value updates immediately because it is treated as urgent state.
- Deferred List Update: Filtering the list happens inside startTransition, treating it as low-priority work.
- Loading Feedback: The isPending flag indicates when the low-priority update is in progress, allowing you to show a loading message.

When to use

- Use useDeferredValue when
  - You are working with derived state or props.
  - Expensive computations (e.g., filtering or sorting) depend on a value that updates frequently.
  - You want to keep the UI responsive while deferring the computation.

- Use useTransition when
  - You are triggering state updates that affect multiple parts of the component tree.
  - The update can be deprioritized, like navigation or displaying large lists.
  - You need progress indicators during the low-priority update.

Key Differences Between useDeferredValue and useTransition

Feature	useDeferredValue	useTransition
Primary Use Case	Defers updates to a value, used in child components.	Defers entire state updates, often in parent components.
Type of Updates	Works with derived state (e.g., props).	Works with state updates initiated by startTransition.
Control Level	Applied to a value (e.g., filteredList).	Applied to an update function (setState).
Scope	Narrow, affects a single value.	Broad, affects multiple state updates.
Feedback Mechanism	No direct feedback like isPending.	Provides isPending to show progress indicators.
Typical Use Cases	Filtering, sorting, rendering derived data.	Navigation, multi-step updates, expensive calculations.