**Performance Optimization Techniques: Updated Table**

| **Optimization Technique** | **When to Use** | **When Not to Use** | **Explanation** |
| --- | --- | --- | --- |
| **React.memo** | When child components receive the same props and don’t need to re-render frequently. | Avoid if components have dynamic values or if the comparison overhead exceeds the cost of re-rendering. | Memoizes components, preventing unnecessary re-renders. |
| **useCallback** | When passing functions as props to child components or using functions in dependencies that don’t need re-creation. | Avoid when function creation is inexpensive or when it adds more complexity without significant performance benefits. | Optimizes function references to avoid re-creation on every render. |
| **useMemo** | When computations or operations are expensive and don’t need to run on every render. | Avoid for cheap calculations where memoization overhead exceeds the performance gain. | Memoizes expensive calculations or object references, preventing unnecessary recomputation. |
| **React.lazy & Suspense** | When you want to load large components or routes only when they are needed, reducing initial load times. | Avoid for small components where lazy-loading adds unnecessary complexity or too many small fallbacks in the UI. | Lazy-loads components, improving initial load time by reducing the initial bundle size. |
| **Code Splitting** | When you have large parts of your app that are not needed on the initial page load. | Avoid for small, critical components that are essential to the first paint. | Splits code into smaller, on-demand chunks, improving load times and interactivity. |
| **Batched Updates** | When multiple state updates are triggered in the same event loop, such as user interactions or API calls. | Batched updates are automatic in React 18, but still avoid excessive state updates in general as they could affect performance even when batched. | Consolidates multiple state updates into a single render to prevent redundant re-renders. |
| **useEffect** | When handling side effects like data fetching, subscriptions, or DOM updates. | Avoid re-running side effects on every render by correctly managing dependencies. | Optimizes side effects by controlling when they run (e.g., on mount or dependency change). |
| **useReducer** | When state logic is complex or when updates depend on the previous state. | Avoid for simple, non-interdependent states where useState is easier to manage. | Optimizes state management by bundling complex logic into a single reducer, improving maintainability and performance for complex states. |
| **Avoid Anonymous Functions** | When functions are used inside JSX or passed as props to child components, causing unnecessary re-renders. | Avoid if the function is only used once and its creation doesn’t affect performance. | Prevents creating new function instances on every render, which can trigger unnecessary re-renders, especially in child components. |
| **useRef** | When you need to persist values across renders without causing a re-render (e.g., DOM refs, timers, previous state). | Avoid for state or data that needs to trigger re-renders; use useState in those cases. | Allows storing mutable values (like DOM references) that don’t affect rendering. |
| **Optimize useEffect** | When managing side effects, ensuring that only necessary effects run when dependencies change. | Avoid adding unnecessary dependencies, as this could trigger unintended re-renders and performance bottlenecks. | Improves performance by reducing unnecessary side effect re-runs and limiting unnecessary renders. |