УДК 004.438.5

Панов Георгий Александрович, магистрант, Российский университет транспорта, Москва, Россия

РАЗРАБОТКА И ЭКСПЕРИМЕНТАЛЬНАЯ ВЕРИФИКАЦИЯ КОМПОНЕНТА ВИРТУАЛИЗИРОВАННОГО СПИСКА НА REACT

Аннотация

В статье представлены проектирование, программная реализация и экспериментальная оценка компонента виртуализированного списка на платформе React. Рассмотрены архитектурные решения, обеспечивающие масштабируемое отображение больших наборов данных: фиксированный пул DOM-элементов, абсолютное позиционирование с использованием CSS-трансформаций, имитация полной высоты списка и оптимизация повторных вычислений средствами React. Проведено экспериментальное сравнение предложенного компонента со стратегией полной загрузки на наборах данных объемом 1 000, 10 000 и 100 000 записей в условиях эмуляции маломощного мобильного устройства. Показано, что виртуализированный список сохраняет практически постоянные показатели по числу DOM-узлов, объему памяти и времени отклика. Для сценария с 100 000 записей получены значения LCP 1,03 с, 140 DOM-узлов и 10,7 МБ памяти по снимку кучи, тогда как полная загрузка потребовала 56,66 с, 300 041 DOM-узел и 478 МБ памяти. Результаты подтверждают высокую эффективность предложенного подхода и его пригодность для высоконагруженных веб-приложений.

Annotation

The paper presents the design, implementation, and experimental evaluation of a virtualized list component built on the React platform. The study describes architectural

solutions that enable scalable visualization of large data sets, including a fixed pool of DOM elements, absolute positioning with CSS transforms, simulation of the full list height, and optimization of repeated calculations using React mechanisms. An experimental comparison is carried out between the proposed component and the full rendering strategy on data sets of 1,000, 10,000, and 100,000 records under low-tier mobile device emulation. The results show that the virtualized list maintains nearly constant characteristics in terms of DOM node count, memory consumption, and response time. For the 100,000-record scenario, the component achieved an LCP of 1.03 s, 140 DOM nodes, and 10.7 MB of heap snapshot memory, whereas full rendering required 56.66 s, 300,041 DOM nodes, and 478 MB of memory. The findings confirm the high efficiency of the proposed approach and its applicability to high-load web applications.

Ключевые слова: React, виртуализированный список, веб-приложения, производительность интерфейса, DOM, экспериментальная верификация.

Keywords: React, virtualized list, web applications, interface performance, DOM, experimental verification.

Текст статьи

Практическая реализация высокопроизводительного отображения больших наборов данных в веб-приложениях требует перехода от теоретических моделей к инженерным решениям, способным подтвердить свою эффективность в реальных условиях исполнения. Если на концептуальном уровне виртуальный рендеринг рассматривается как способ обеспечения условий $M_{DOM}(n) = O(1)$ и $T(n) = O(1)$, то на прикладном уровне важно показать, каким образом данные требования достигаются в конкретной компонентной архитектуре и как они проявляются в измеряемых пользовательских метриках.

Целью настоящей работы является разработка компонента виртуализированного списка на React и его экспериментальная верификация в сравнении с традиционной стратегией полной загрузки данных. В качестве объекта исследования рассматривается интерфейсный компонент отображения линейного списка записей большой размерности, а предметом исследования выступают его архитектурные и производительные характеристики.

Разработанный компонент VirtualizedList основан на четырех принципах. Во-первых, используется фиксированный пул DOM-элементов, число которых определяется высотой видимой области и буфером предварительной отрисовки overscan. При высоте контейнера 500 px и высоте строки 35 px даже с учетом буфера одновременно требуется лишь ограниченное число визуальных контейнеров. Формально верхнюю границу числа отрисованных строк можно представить выражением:

$$N_{visible} = \lceil H_{container}/h_{item} \rceil + 2 \cdot overscan.$$

При $H_{container} = 500$, $h_{item} = 35$ и $overscan = 5$ получаем:

$$N_{visible} = \lceil 500/35 \rceil + 10 = 15 + 10 = 25.$$

Следовательно, даже при увеличении общего числа записей n до 100 000 и более количество одновременно активных DOM-элементов остается ограниченным малой константой, что и обеспечивает архитектурную масштабируемость по части визуализации.

Во-вторых, позиционирование элементов реализуется абсолютно, а вертикальное смещение задается с помощью CSS-свойства transform: translateY(). Для i-й строки ее положение определяется соотношением:

$$Y_i = i \cdot h_{item}.$$

Такой способ размещения снижает количество дорогостоящих операций layout/reflow и позволяет браузеру эффективнее использовать графический конвейер. По сравнению с подходом на основе постоянного изменения top это уменьшает нагрузку на пересчет геометрии и улучшает плавность прокрутки на устройствах с ограниченной вычислительной мощностью.

В-третьих, для сохранения корректной механики прокрутки применяется элемент-заполнитель, формирующий полную высоту списка:

$$H_{total} = n \cdot h_{item}.$$

Это обеспечивает пользователю восприятие непрерывного списка, хотя в DOM фактически присутствует лишь ограниченное окно видимых элементов. В-четвертых, дополнительные оптимизации реализованы средствами useMemo, useCallback и React.memo, что уменьшает число лишних перерасчетов и повторных перерисовок дочерних компонентов.

Вычисление границ видимого диапазона осуществляется по формулам:

$$visibleStart = \lfloor scrollTop / h_{item} \rfloor$$

$$visibleEnd = \lceil (scrollTop + H_{container}) / h_{item} \rceil - 1$$

$$startIndex = \max(0, visibleStart - overscan)$$

$$endIndex = \min(n - 1, visibleEnd + overscan)$$

Соответствующий фрагмент реализации имеет следующий вид:

```
const visibleStart = Math.floor(scrollTop / itemHeight);
const visibleEnd = Math.ceil((scrollTop + containerHeight) / itemHeight) - 1;
const startIndex = Math.max(0, visibleStart - overscan);
const endIndex = Math.min(numberOfItems - 1, visibleEnd + overscan);
```

Генерация только видимых элементов выполняется в пределах вычисленного диапазона:

```
const visibleItems = [];
for (let i = startIndex; i <= endIndex; i++) {
  visibleItems.push(
    <div
      key={i}
      style={{
        position: "absolute",
        top: 0,
        left: 0,
        right: 0,
        height: `${itemHeight}px`,
        transform: `translateY(${i * itemHeight}px)`,
      }}
    >
      <ListItem index={i} />
    </div>,
  );
}
```

Архитектурно важным является то, что алгоритм не выполняет обход полного массива для каждого кадра прокрутки. На каждом событии используются только арифметические операции и граничные проверки, количество которых не зависит от n. За счет этого временная сложность шага обновления визуального диапазона остается постоянной относительно объема данных.

Для экспериментальной проверки были подготовлены два тестовых приложения. Приложение A реализует стратегию полной загрузки и рендерит все записи сразу. Приложение B использует разработанный компонент VirtualizedList. Измерения выполнялись в браузере Google Chrome 122 в режиме эмуляции маломощного

мобильного устройства с коэффициентом замедления CPU 11x (preset Low-tier mobile device). Каждый тест запускался серией повторов с очисткой кэша и последующей перезагрузкой страницы; в анализ включались медианные значения метрик.

В качестве основных метрик выбраны: LCP (Largest Contentful Paint), количество DOM-узлов, пиковый размер JS Heap, общий объем памяти по heap snapshot, а также суммарное время выполнения главного потока (Total). Такой набор позволяет одновременно оценить пользовательскую скорость отображения контента, структурную нагрузку на DOM, память и вычислительные затраты JavaScript/рендеринга.

Эксперимент проводился на наборах данных размером 1 000, 10 000 и 100 000 записей. Наиболее показательным является сценарий с максимальным объемом, поскольку именно он позволяет выявить различия в масштабируемости архитектур.

Метрика	Полная загрузка	Виртуализация	Улучшение
LCP, с	56,66	1,03	98,2%
DOM-узлы, шт.	300 041	140	99,95%
JS Heap, МБ	212	6,5	96,9%
Память (снимок кучи), МБ	478	10,7	97,8%
Total, мс	62 083	1 408	97,7%

Полученные результаты демонстрируют качественное различие между сравниваемыми стратегиями. При 100 000 записей полная загрузка формирует более 300 000 DOM-узлов, потребляет 478 МБ памяти и требует более 62 секунд суммарного времени работы главного потока. Это делает интерфейс практически непригодным для интерактивного использования. Виртуализированный список,

напротив, сохраняет стабильные показатели: около 140 DOM-узлов, 10,7 МБ памяти и LCP 1,03 с, что соответствует “хорошей” зоне Core Web Vitals.

Для количественного описания выигрыша можно использовать относительное улучшение:

$$I = \frac{X_{full} - X_{virtual}}{X_{full}} \cdot 100\%.$$

Например, для количества DOM-узлов:

$$I_{DOM} = \frac{300041 - 140}{300041} \cdot 100\% \approx 99{,}95\%.$$

Для времени LCP:

$$I_{LCP} = \frac{56{,}66 - 1{,}03}{56{,}66} \cdot 100\% \approx 98{,}2\%.$$

Даже на меньших наборах данных виртуализация демонстрирует устойчивое преимущество: сокращение числа DOM-узлов на 95-99%, уменьшение потребления памяти на 36-82% и ускорение LCP на 49-82%. Это подтверждает, что выигрыш не является локальным эффектом только для экстремального сценария, а представляет собой устойчивое свойство выбранной архитектуры.

Отдельно следует отметить интерпретацию результата по памяти. В эксперименте и при полной загрузке, и при виртуализации данные полностью доступны клиенту; различие формируется в первую очередь за счет сокращения рендер-части памяти, то есть числа одновременно присутствующих DOM-узлов и связанных с ними структур движка браузера. Это согласуется с постановкой задачи: оптимизируется не хранение бизнес-данных как таковых, а стратегия их визуального представления.

Экспериментальные данные согласуются с теоретическими предпосылками. Так как число отображаемых узлов ограничено величиной $N_{visible}$, архитектура компонента сохраняет асимптотически постоянные затраты на визуализацию независимо от n. При этом полная загрузка масштабируется линейно, а в ряде аспектов приводит и к каскадному росту внутренних затрат браузера, связанных с перерасчетом layout, paint и обслуживанием больших DOM-структур. Следовательно, предложенное решение подтверждает практическую реализуемость модели $M_{DOM}(n) = O(1)$ и $T(n) = O(1)$ в условиях современного веб-стека.

К ограничениям проведенной верификации можно отнести использование фиксированной высоты строки и синтетических данных равномерной структуры. Для промышленных сценариев с переменной высотой элементов, вложенными интерактивными блоками и асинхронным обновлением содержимого требуется дополнительная адаптация: кэширование измерений высоты, стабилизация ключей, контроль частоты событий прокрутки и более сложная стратегия восстановления позиции. Тем не менее базовая архитектура виртуализированного окна сохраняет применимость и в этих условиях.

Таким образом, разработанный компонент виртуализированного списка на React не только реализует теоретические принципы виртуального рендеринга, но и демонстрирует измеримое превосходство по ключевым метрикам производительности. Это позволяет рекомендовать данный подход для корпоративных систем, аналитических панелей, журналов событий, справочников и иных интерфейсов, в которых требуется отображение больших массивов данных без потери отзывчивости и с сохранением непрерывного пользовательского опыта.

Литература

1. Наибольшая отрисовка контента (LCP) [Электронный ресурс] // web.dev : [сайт]. URL: https://web.dev/articles/lcp (дата обращения: 22.04.2026).
2. Основные интернет-показатели (Core Web Vitals) [Электронный ресурс] // Google Developers : [сайт]. URL: https://developers.google.com/search/docs/appearance/core-web-vitals (дата обращения: 22.04.2026).
3. MDN Web Docs. Введение в DOM (Document Object Model) [Электронный ресурс] // MDN Web Docs : [сайт]. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (дата обращения: 22.04.2026).
4. Google Developers. Производительность рендеринга [Электронный ресурс] // Web Fundamentals : [сайт]. URL: https://developers.google.com/web/fundamentals/performance/rendering (дата обращения: 22.04.2026).
5. Chrome for Developers. Проблемы с памятью [Электронный ресурс] // Chrome DevTools : [сайт]. URL: https://developer.chrome.com/docs/devtools/memory-problems/ (дата обращения: 22.04.2026).
6. React Team. Документация React: рендеринг списков и оптимизация обновлений интерфейса [Электронный ресурс] // React : [сайт]. URL: https://react.dev/learn/rendering-lists (дата обращения: 22.04.2026).
7. React Team. Справочник API хуков [Электронный ресурс] // React : [сайт]. URL: https://react.dev/reference/react (дата обращения: 22.04.2026).
8. Google Developers. Виртуализация длинных списков в веб-приложениях [Электронный ресурс] // web.dev : [сайт]. URL:

https://web.dev/articles/virtualize-long-lists-react-window (дата обращения: 22.04.2026).

9. MDN Web Docs. CSS-свойство transform [Электронный ресурс] // MDN Web Docs : [сайт]. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/transform (дата обращения: 22.04.2026).
10. Chrome for Developers. Панель Performance в Chrome DevTools [Электронный ресурс] // Chrome for Developers : [сайт]. URL: https://developer.chrome.com/docs/devtools/performance/ (дата обращения: 22.04.2026).
11. Chrome for Developers. Инспектор памяти (Memory Inspector) [Электронный ресурс] // Chrome for Developers : [сайт]. URL: https://developer.chrome.com/docs/devtools/memory-inspector/ (дата обращения: 22.04.2026).
12. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 4-е изд. М.: Вильямс, 2020. 1328 с.

Literature

1. Largest Contentful Paint (LCP) [Electronic resource] // web.dev. URL: https://web.dev/articles/lcp (accessed: 22.04.2026).
2. Core Web Vitals [Electronic resource] // Google Developers. URL: https://developers.google.com/search/docs/appearance/core-web-vitals (accessed: 22.04.2026).
3. MDN Web Docs. Introduction to the Document Object Model (DOM) [Electronic resource] // MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (accessed: 22.04.2026).
4. Google Developers. Rendering performance [Electronic resource] // Web Fundamentals. URL:

https://developers.google.com/web/fundamentals/performance/rendering (accessed: 22.04.2026).

5. Chrome for Developers. Memory problems [Electronic resource] // Chrome DevTools. URL: https://developer.chrome.com/docs/devtools/memory-problems/ (accessed: 22.04.2026).
6. React Team. React documentation: rendering lists and optimizing UI updates [Electronic resource] // React. URL: https://react.dev/learn/rendering-lists (accessed: 22.04.2026).
7. React Team. Hooks API reference [Electronic resource] // React. URL: https://react.dev/reference/react (accessed: 22.04.2026).
8. Google Developers. Virtualize large lists in web applications [Electronic resource] // web.dev. URL: https://web.dev/articles/virtualize-long-lists-react-window (accessed: 22.04.2026).
9. MDN Web Docs. CSS transform property [Electronic resource] // MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/transform (accessed: 22.04.2026).
10. Chrome for Developers. Performance panel in Chrome DevTools [Electronic resource] // Chrome for Developers. URL: https://developer.chrome.com/docs/devtools/performance/ (accessed: 22.04.2026).
11. Chrome for Developers. Memory Inspector [Electronic resource] // Chrome for Developers. URL: https://developer.chrome.com/docs/devtools/memory-inspector/ (accessed: 22.04.2026).
12. Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms. 4th ed. Moscow: Williams, 2020. 1328 p.