УДК 004.4
DOI 10.17513/snt.40303

ОЦЕНКА АЛГОРИТМОВ DOCUMENT OBJECT MODEL, SIMPLE API FOR XML И STREAMING API FOR XML ДЛЯ РАБОТЫ С XML В ВЫСОКОНАГРУЖЕННЫХ ПРИЛОЖЕНИЯХ

Золотухина Д.Ю.

ФГБОУ ВО «Воронежский государственный университет», Воронеж, e-mail: dar.zolott@gmail.com

Цель работы заключается в сравнительном анализе производительности алгоритмов Document Object Model, Simple API for XML и Streaming API for XML при выполнении задач чтения, записи и частичного чтения XML-файлов различного объема. Для проведения экспериментов была разработана модульная программа на языке Java, где каждый алгоритм реализован в отдельном модуле. В качестве данных использовались синтетически сгенерированные XML-файлы с типовой структурой, имитирующей транзакции. Эксперименты проводились с замером ключевых метрик, таких как время выполнения и потребление памяти. В ходе исследования были выявлены сильные и слабые стороны каждого алгоритма. Simple API for XML продемонстрировал наилучшую производительность и минимальное потребление памяти, что делает его подходящим для обработки больших объемов данных. Streaming API for XML обеспечил баланс между производительностью и удобством реализации, предоставляя более простой доступ к данным. Document Object Model, несмотря на удобный интерфейс, оказался слишком ресурсоемким и может быть использован только для небольших объемов данных. Результаты работы подчеркивают важность выбора алгоритма обработки XML в зависимости от специфики задач. Simple API for XML и Streaming API for XML являются предпочтительными для высоконагруженных систем, требующих минимизации затрат на ресурсы. Дальнейшие исследования могут быть направлены на изучение альтернативных форматов данных и их интеграции с современными архитектурными подходами.

Ключевые слова: обработка XML, Document Object Model, Simple API for XML, Streaming API for XML, производительность алгоритмов, Java, высоконагруженные системы

EVALUATION OF DOCUMENT OBJECT MODEL, SIMPLE API FOR XML AND STREAMING API FOR XML ALGORITHMS FOR XML PROCESSING IN HIGH-LOAD APPLICATIONS

Zolotukhina D.Yu.

Voronezh State University, Voronezh, e-mail: dar.zolott@gmail.com

The aim of the study is to conduct a comparative analysis of the performance of Document Object Model, Simple API for XML and Streaming API for XML algorithms in solving tasks related to reading, writing, and partial reading of XML files of various sizes. A modular Java program was developed for the experiments, with each algorithm implemented in a separate module. Synthetic XML files with a typical structure simulating transactions were used as data. The experiments measured key metrics such as execution time, memory consumption, and qualitative observations. The study identified the strengths and weaknesses of each algorithm. Simple API for XML demonstrated the best performance and minimal memory consumption, making it suitable for processing large volumes of data. Streaming API for XML provided a balance between performance and ease of implementation, offering more convenient access to data. Document Object Mode, despite its convenient interface, was found to be too resource-intensive and suitable only for small volumes of data. The results emphasize the importance of choosing an XML processing algorithm based on task specifics. Simple API for XML and Streaming API for XML are preferred for high-load systems requiring resource minimization. Future research could explore alternative data formats and their integration with modern architectural approaches.

Keywords: XML processing, Document Object Model, Simple API for XML, Streaming API for XML, algorithm performance, Java, high-load systems

Введение

Обработка данных в формате XML остается важной задачей для многих приложений, включая системы обмена данными, интеграционные платформы и финансовые сервисы. Эффективность работы с XML напрямую зависит от выбора алгоритма обработки, который должен обеспечивать высокую производительность и оптимальное использование ресурсов. Наиболее популярными алгоритмами работы с XML являются DOM, SAX и StAX, каждый из которых имеет свои собственные характеристики и области применения [1].

DOM (Document Object Model) предоставляет удобный интерфейс для работы с XML, позволяя загружать весь документ в память и проводить манипуляции с его структурой [2]. Однако это влечет за собой значительные затраты памяти и времени, особенно при обработке больших файлов [3]. SAX (Simple API for XML) использует событийно-ориентированный подход, что делает его более производительным

и менее ресурсоемким, но усложняет реализацию функций, требующих произвольного доступа к элементам [4]. StAX (Streaming API for XML) сочетает преимущества потоковой обработки и удобного доступа к элементам, предоставляя баланс между производительностью и гибкостью [5].

Целью исследования является сравнительный анализ производительности алгоритмов DOM, SAX и StAX при выполнении задач чтения, записи и частичного чтения XML-файлов разного объема с учетом времени выполнения, потребления оперативной памяти и удобства реализации.

Материалы и методы исследования

Для проведения исследования была разработана программа на языке Java, предназначенная для тестирования производительности алгоритмов обработки XML-файлов: DOM, SAX и StAX. Программа была спроектирована как модульное приложение, разделенное на три ключевых блока, каждый из которых реализовывал одну из задач. Программа включала три модуля: чтение XML-файлов, запись XML-файлов и чтение частичных данных. Каждый модуль был реализован как независимый класс для удобства тестирования и сравнения характеристик. Структура тестовых данных была однотипна для всех задач. XML-файлы содержали элементы <transaction>, включавшие в себя следующие поля: <id> – уникальный идентификатор записи, <amount> – сумма транзакции от 1 до 10 000, <timestamp> – дата и время транзакции в формате ISO 8601, <description> – текстовое описание содержимого транзакции.

Пример структуры XML:

```
<transactions>
    <transaction>
        <id>1</id>
        <amount>123.45</amount>
        <timestamp>2024-01-01T10:00:00</timestamp>
        <description>Payment</description>
    </transaction>
</transactions>
```

Для экспериментов были сгенерированы файлы трех категорий объемов: 10 MB (10000 записей), 100 MB (100000 записей) и 1 GB (1000000 записей). Генерация файлов была реализована с использованием Java-программы, что обеспечило единообразие структуры данных для всех экспериментов.

В рамках каждого эксперимента замерялись следующие ключевые параметры:

1. Время обработки (мс): Продолжительность выполнения задачи (чтения, записи, частичного чтения).

2. Потребление памяти (MB): Пиковый объем оперативной памяти, использованный во время выполнения задачи.

Среда, в которой проводилось тестирование, имела следующие характеристики:

− Процессор: Intel Core i7-12700K (12 ядер, 20 потоков, 3.6 GHz);

− Оперативная память: 32 GB DDR4;

− Накопитель: NVMe SSD 1 TB;

− Операционная система: Ubuntu 22.04;

− JVM: OpenJDK 17.

Для измерения производительности использовались встроенные таймеры Java (System.currentTimeMillis()) для замера времени выполнения, VisualVM для анализа потребления памяти и мониторинга работы JVM, а также Java Flight Recorder для записи временных характеристик выполнения программы.

Схема архитектуры тестовой программы представлена на рис. 1.

Данные метрики были выбраны на основании их практической ценности для анализа производительности алгоритмов обработки XML-файлов. Каждая задача запускалась трижды для минимизации случайных ошибок, а результаты усреднялись.

Результаты исследования и их обсуждение

Результаты экспериментов по анализу производительности алгоритмов DOM, SAX и StAX представлены в таблице.

Таблица демонстрирует, что алгоритм DOM показал наибольшее время обработки и потребление памяти во всех экспериментах. Это связано с полной загрузкой XML-файла в оперативную память и построением дерева документа [6, с. 57–58]. Например, при чтении файла объемом 1 GB время обработки составило 58000 мс, а потребление памяти достигло 3600 MB. Эти показатели делают DOM неподходящим выбором для работы с большими объемами данных, особенно в условиях ограниченных вычислительных ресурсов. Однако его способность к удобной манипуляции структурой документа остается полезной для задач, требующих полного доступа к содержимому XML.

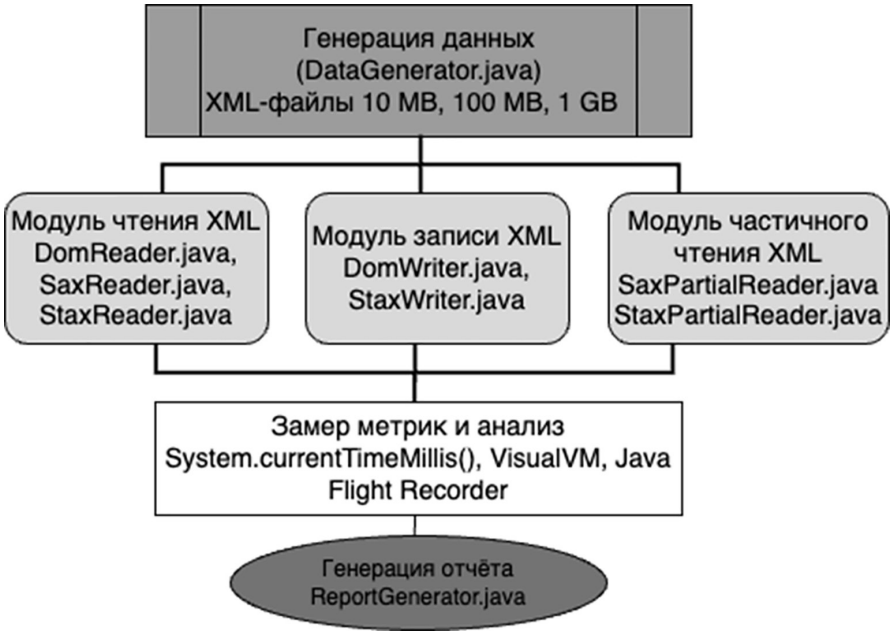

Рис. 1. Схема архитектуры программы
Источник: составлено авторами

Сравнительный анализ производительности алгоритмов DOM, SAX и StAX

Эксперимент	Объем файла	Алгоритм	Время обработки (мс)	Потребление памяти (MB)
Чтение	10 MB	DOM	480	45
	10 MB	SAX	220	15
	10 MB	StAX	260	20
	100 MB	DOM	4850	380
	100 MB	SAX	1300	50
	100 MB	StAX	1700	65
	1 GB	DOM	58000	3600
	1 GB	SAX	9800	110
	1 GB	StAX	11200	140
Запись	10 MB	DOM	950	50
	10 MB	StAX	520	28
	100 MB	DOM	9400	420
	100 MB	StAX	5200	140
	1 GB	DOM	105000	3800
	1 GB	StAX	58700	520
Частичное чтение данных	10 MB	SAX	210	14
	10 MB	StAX	250	18
	100 MB	SAX	1220	45
	100 MB	StAX	1550	60
	1 GB	SAX	900	100
	1 GB	StAX	10800	130

Источник: составлено авторами.

SAX продемонстрировал лучшие результаты по времени обработки и минимальному потреблению памяти благодаря потоковому подходу, который обрабатывает XML-документ последовательно, без необходимости хранения полной структуры в памяти [7]. Например, для задачи чтения файла объемом 1 GB время выполнения составило 9800 мс при потреблении памяти всего 110 MB. Это делает SAX особенно эффективным для задач, связанных с обработкой больших объемов данных в условиях высокой нагрузки. Однако, несмотря на преимущества, сложности разработки кода на SAX, включая необходимость обработки событий и отслеживания текущего состояния, могут стать препятствием для его использования в более сложных сценариях.

StAX занял промежуточное положение между DOM и SAX, обеспечивая баланс между производительностью и удобством реализации. Для задачи чтения файла объемом 1 GB время выполнения составило 11200 мс, а потребление памяти достигло 140 MB. В отличие от SAX, StAX предлагает более удобный программный интерфейс, что делает его подходящим выбором для задач, где требуется потоковая обработка с доступом к отдельным элементам XML. При этом его производительность остается близкой к SAX, хотя и несколько ниже из-за дополнительных накладных расходов на управление состоянием.

При записи XML-файлов StAX продемонстрировал лучшее соотношение времени обработки и потребления памяти. Например, для записи файла объемом 1 GB время выполнения составило 58700 мс, что почти вдвое меньше, чем у DOM, который показал результат в 105000 мс. Это связано с тем, что StAX позволяет записывать данные по мере их генерации, избегая необходимости построения полной структуры документа в памяти. DOM, напротив, требует построения дерева документа перед записью, что приводит к значительным накладным расходам [8].

Во время выполнения задачи частичного чтения наилучший результат продемонстрировала технология SAX за счет наименьшего потребления памяти и времени работы. При использовании данного алгоритма удалось обработать файл, размер которого 1 Гб, за 9700 мс, что немного лучше результата StAX, который составил 10800 мс.

На рисунке 2 изображен график зависимости времени, затраченного на чтение файла, от его объема. SAX осуществляет самую быструю обработку файлов каждого размера, а время чтения документа алгоритмом DOM растет экспоненциально по мере увеличения объема данных. StAX показывает результат немного хуже, чем у SAX, но их рост является пропорциональным.

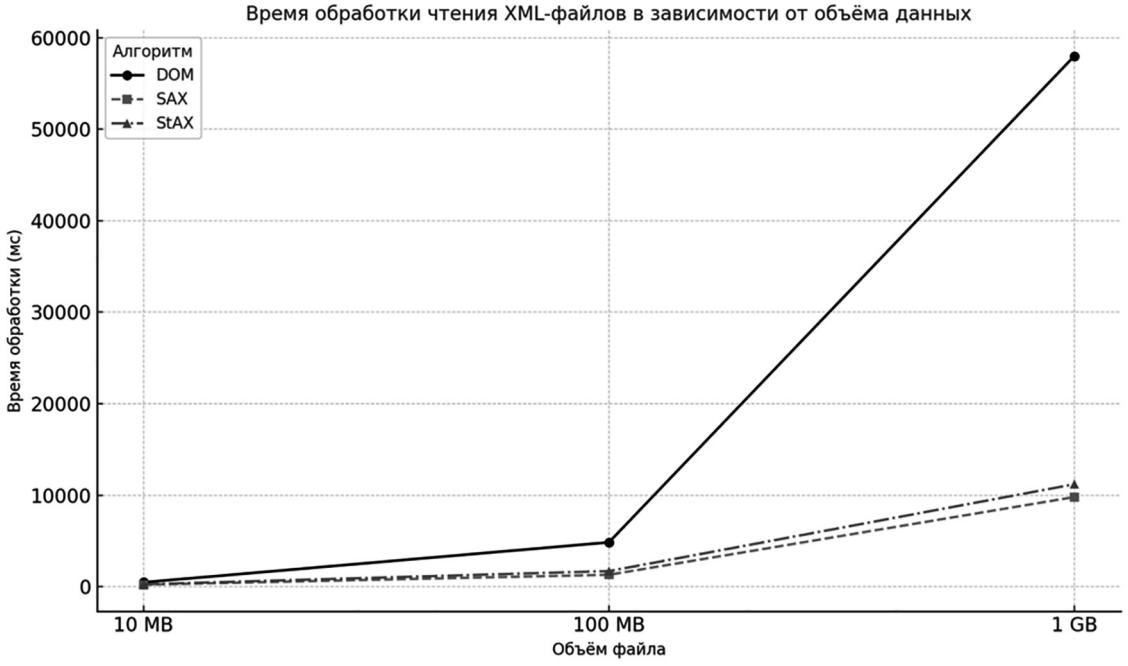

Рис. 2. Время обработки чтения XML-файла в зависимости от объема данных
Источник: составлено авторами

Обсуждая полученные результаты, важно рассмотреть дополнительные практические рекомендации, которые могут помочь инженерам выбрать наилучший алгоритм для решения конкретной задачи. DOM будет являться предпочтительной технологией, если при работе с файлом требуются многократные возвращения к уже прочитанным узлам XML. Также это актуально для задач, связанных с генерацией XML-документа, если в них необходимо постоянно возвращаться и вносить правки в уже сгенерированные узлы. DOM в этом случае обеспечивает эффективную работу благодаря тому, что вся структура доступна в памяти в виде дерева. Однако важно учитывать, что при работе программы в условиях ограниченной оперативной памяти использование DOM может привести к частому срабатыванию сборщика мусора (Garbage Collector). Данный процесс может дополнительно увеличить время обработки.

В высоконагруженных распределенных системах должна быть возможность организовать структуру приложения таким образом, чтобы узлы (instance) могли параллельно обрабатывать разные фрагменты документа или несколько файлов одновременно [9; 10]. В этом случае подходящими технологиями для обработки будут являться SAX и StAX, которые позволяют читать поток сразу, без необходимости загрузки полной структуры XML в память.

При необходимости обработки XML со сложной структурой, где логика предполагает частую фильтрацию и агрегацию данных или одновременное чтение разных частей документа, нужно проанализировать, какой подход сделает разработку программы более удобной и быстрой: событийный у SAX или поток-курсор у StAX. Событийный подход предполагает вызов обработчиков при наступлении определенных событий (начало элемента, конец элемента, текст и т.д.), поэтому программисту приходится вручную сохранять состояние для выполнения более сложной логики. Потоково-курсорный подход работает как «указатель», последовательно перемещающийся по элементам XML. Это упрощает доступ к нужным данным и управление потоком обработки, однако может потребовать дополнительного кода для выборочного чтения и пропуска определенных сегментов документа.

Независимо от выбранного алгоритма, перед развертыванием в промышленной среде рекомендуется проводить нагрузочное тестирование (load testing) и профилирование (profiling). Такие инструменты, как VisualVM и Java Flight Recorder, упомянутые в статье, позволяют выявить «узкие места» и оптимизировать код.

Заключение

Анализируя результаты проведенного исследования, можно сделать вывод о том, что выбор наилучшей технологии для работы с XML-файлами зависит от требований системы, объема обрабатываемых данных и специфики задачи. SAX является наиболее эффективным алгоритмом, так как ему требуется наименьшее время и объем памяти для проведения операций над файлами любых размеров. В то же время SAX показывает меньшую производительность, но выигрывает за счет простоты и удобства интерфейса, что делает его подходящим выбором для задач, включающих в себя сложные запросы с несколькими условиями. DOM подходит только для работы с небольшим размером данных, но является полезным инструментом для построения полного дерева документа в памяти и управления его структурой.

Одной из ключевых особенностей, выявленных в ходе экспериментов, стала важность оценки потребления памяти наряду с производительностью. В условиях современных высоконагруженных систем, где ресурсы серверов часто лимитированы, использование алгоритмов с минимальными требованиями к памяти становится приоритетным. Это делает SAX предпочтительным выбором в сценариях, где работа с большими объемами данных происходит в условиях ограниченной инфраструктуры.

Полученные результаты имеют прикладное значение для разработки высоконагруженных приложений, требующих обработки XML-файлов. Применение этих алгоритмов может быть эффективно адаптировано для различных систем, включая серверные платформы, интеграционные решения и системы обмена данными. Перспективы дальнейших исследований могут включать анализ новых подходов, таких как использование бинарных форматов XML и их влияние на производительность, а также исследование интеграции алгоритмов с современными архитектурными подходами, включая микросервисы и контейнеризацию.

Список литературы

1. Ali M., Khan M.A. Enhancing XML Data Parsing and Querying Performance on Multi-Core Architectures // Journal of Statistics, Computing and Interdisciplinary Research. 2024. № 6–1. P. 75–89. DOI: 10.52700/scir.v6i1.158.

2. Schweinsberg K., Wegner L. Advantages of complex SQL types in storing XML documents // Future Generation Computer Systems. 2017. T. 68. P. 500–507. DOI: 10.1016/j.future.2016.02.013.

3. Ali M., Khan M.A. Performance enhancement of XML parsing using regression and parallelism // Computer systems science and engineering. 2024. № 48–2. P. 287–303. DOI: 10.32604/csse.2023.043010.

4. Deshmukh V.M., Bamnote G.R. An Empirical Study of XML Parsers across Applications // 2015 International Conference on Computing Communication Control and Automation. IEEE, 2015. P. 396–401. DOI: 10.1109/ICCUBEA.2015.83.

5. Зубов М.В., Пустыгин А.Н. Использование абстрактного цифрового автомата для получения универсального промежуточного представления исходного кода программ // Вестник Астраханского государственного технического университета. Серия: Управление, вычислительная техника и информатика. 2015. №. 4. С. 57–65. URL: https://vestnik.astu.org/ru/nauka/article/32480/view (дата обращения: 20.12.2024).

6. Friesen J. Java XML and JSON. New York: Apress, 2016. 308 p.

7. Wu D.Y., Chau K.T., Wang J.Y., Pan C.T. A comparative study on performance of XML parser apis (DOM and SAX) in parsing efficiency // Proceedings of the 3rd International Conference on Cryptography, Security and Privacy. 2019. P. 88–92. DOI: 10.1145/3309074.330912.

8. Белых Е.А., Гольчевский Ю.В. Анализ существующих способов решения задачи генерации сложных электронных документов на основе шаблонов // Информационно-технологический вестник. 2024. № 4 (42). С. 66–76.

9. Матчин В.Т., Плотников С.Б., Цветков В.Я. Работа с информационными ресурсами в высоконагруженных приложениях // Образовательные ресурсы и технологии. 2020. № 4 (33). С. 62–72. DOI: 10.21777/2500-2112-2020-4-62-72.

10. Рудометкин В.А. Повышение отказоустойчивости высоконагруженных систем // Информационно-технологический вестник. 2020. № 3 (25). С. 118–123. DOI: 10.21499/2409-1650-2020-25-3-118-123.