DOI: https://doi.org/10.26642/ten-2026-1(97)-210-216
УДК 004.421.2:004.912:004.75

Д.В. Зважій, аспірант, ст. викладач
О.М. Малій, студентка
Національний університет «Києво-Могилянська академія»

Побудова розподілених суфіксних дерев

У статті розглядається задача паралельної побудови суфіксних дерев над великими текстовими масивами без використання кластерної інфраструктури. Проаналізовано еволюцію відповідних алгоритмів – від однопотокових методів Вайнера, МакКрейта та Укконена до розподілених підходів Wavefront, ER та DGST. Виявлено ключове обмеження алгоритму DGST, реалізованого на платформі Apache Spark: суттєві накладні витрати на серіалізацію даних та I/O-звернення до розподіленої файлової системи HDFS.

Запропоновано архітектуру адаптації алгоритму DGST до середовища Node.js на основі модуля worker_threads та механізму SharedArrayBuffer. Структурна трансформація забезпечує відображення компонентів Spark на елементи Node.js: керуючий вузол – основний потік, виконавчий вузол – воркерний потік, HDFS – SharedArrayBuffer. Завдяки принципу нульового копіювання (Zero-Copy) та кешуванню декодованого вмісту буфера усуваються накладні витрати на серіалізацію, характерні для Spark, а архітектура переходить від I/O-bound до memory-bound моделі. Описано чотири алгоритмічні фази адаптованого методу: ініціалізація та попередня обробка, ітеративне розбиття на S-префікси, фаза Map з побудовою локальних частотних префіксних дерев, а також Shuffle/Reduce з балансуванням навантаження на основі алгоритмів Bin Packing та Number Partitioning.

Експериментальне дослідження на текстовому масиві обсягом до 2 млн рядків підтвердило приблизно лінійне зростання часу обробки відносно розміру датасету та сублінійне зростання споживання пам'яті: при чотириразовому збільшенні обсягу даних показник MemoryLimit зріс лише у 3,6 раза. Пропускна здатність системи на максимальному обсязі склала 77 132 рядки/с.

Ключові слова: *суфіксне дерево; DGST; Node.js; worker_threads; SharedArrayBuffer; паралельні обчислення; MapReduce; in-memory обробка.*

Вступ. У сучасних умовах розвитку інформаційних технологій обробка великих текстових масивів є однією з ключових проблем у галузі комп'ютерних наук. Потреба в ефективних алгоритмах для роботи з рядковими даними зумовила появу спеціалізованих структур даних, відомих як суфіксні дерева. Зокрема, у 1973 році Вайнер запропонував перший лінійний алгоритм [1], який згодом було удосконалено та розвинено в роботах видатних науковців МакКрейта [2] та Укконена [3].

Суфіксне дерево стало де-факто стандартом для розв'язання широкого класу задач обробки рядків завдяки можливості виконувати фундаментальні операції за лінійний час. Воно ефективно застосовується для пошуку збігів, підрядків, знаходження найдовших спільних префіксів і паліндромів. Окрім цього, суфіксні дерева активно використовуються в біоінформатиці для аналізу людського геному та виявлення закономірностей в ДНК.

Новим напрямком оптимізації стала паралельна побудова суфіксних дерев на розподілених платформах. Алгоритм DGST (Distributed Generalized Suffix Tree) [4], реалізований на базі Apache Spark, є найбільш показовим представником цього підходу. Він застосовує підхід MapReduce для розподілу обчислень між виконавчими вузлами кластера. Проте розгортання та підтримка такої інфраструктури пов'язані з суттєвими накладними витратами на міжкластерну комунікацію, серіалізацію даних та I/O-звернення до розподіленої файлової системи HDFS (Hadoop Distributed File System), що обмежує застосування підходу поза специфічними умовами.

Для задач, у яких використання та підтримка кластерів є надлишковими, доцільним є перенесення логіки розподілених обчислень обробки даних на серверну архітектуру з використанням багатопотоковості в межах одного процесу. Починаючи з версії v10, середовище виконання Node.js підтримує модуль worker_threads (воркерні потоки), що дає змогу створювати ізольовані потоки для паралелізації CPU-інтенсивних обчислень. В комплексі зі структурою SharedArrayBuffer, яка реалізує модель спільної пам'яті з нульовим копіюванням даних, ці інструменти надають технологічний фундамент для адаптації алгоритму DGST.

Об'єктом дослідження цієї роботи є процес паралельної побудови суфіксних дерев над великими текстовими масивами, а предметом – адаптація алгоритму DGST до серверної архітектури з використанням серверних воркерних потоків та буферів масивів спільного використання (SharedArrayBuffer).

Метою роботи є реалізація та дослідження ефективності алгоритму побудови суфіксних дерев DGST у середовищі Node.js із застосуванням підходу обробки даних в оперативній пам'яті для виконання

ресурсоємних обчислень, а також відтворення моделі MapReduce без потреби у використанні кластерної інфраструктури.

Актуальність обраної тематики зумовлена потребою у практичних методах паралельної обробки великих текстових даних, які забезпечують високу продуктивність без ускладнення програмної та інфраструктурної архітектури.

Вихідний код реалізації доступний у відкритому репозиторії [5].

Аналіз існуючих досліджень. Паралельна побудова суфіксних дерев над великими текстовими масивами є предметом активних досліджень протягом останніх чотирьох десятиліть. Алгоритм Вайнера (1973) став першим в історії методом побудови суфіксних дерев за лінійний час O(n) для алфавітів фіксованого розміру. Однак, його практичне застосування обмежувалося надмірним споживанням пам'яті: кожен внутрішній вузол дерева зберігав допоміжні вектори, розмір яких був пропорційний потужності алфавіту.

Як спрощення та оптимізація підходу Вайнера у 1976 році Маккрейтом був реалізований більш елегантний підхід. На відміну від попередника, алгоритм Маккрейта використав зворотній обхід суфіксів від найдовшого до коротшого із збереженням лише суфіксних посилань замість повних векторів. Подальшу оптимізацію по пам'яті у 1995 році запропонував Укконен, чий алгоритм зберіг характерну лінійну часову складність, але з принциповою зміною парадигми побудови. Якщо методи Вайнера та Маккрейта вимагали повного знання вхідного рядка заздалегідь, то алгоритм Укконена дозволив обробляти вхідні дані «на льоту». Це відкрило можливість потокової обробки даних і заклало основу для подальших досліджень побудови суфіксних дерев.

Не дивлячись на застосовані оптимізації, ключовим недоліком усіх трьох методів залишалася модель обробки даних в оперативній пам'яті. Класичне суфіксне дерево дерево займає у 10–20 разів більше місця, ніж сам рядок, що унеможливлює обробку десятків гігабайтів в оперативній пам'яті в межах одного процесу.

Суттєві обмеження обсягу пам'яті зумовили розробку алгоритмів із використанням зовнішньої пам'яті та паралельних підходів. Першим практично реалізованим паралельним алгоритмом став Wavefront (2009) [6]. Алгоритм розбиває множину суфіксів на партиції, згруповані по спільному префіксу, і для кожної з них ізольовано будує піддерево. Завершальний етап зводиться до злиття часткових піддерев у глобальну структуру. Хоча алгоритм продемонстрував на той час революційну здатність обробити увесь людський геном за 15 хвилин на суперкомп'ютері IBM BlueGene/L з 1024 процесорами, етап злиття все ще потребував збереження двох повних копій дерева у пам'яті виконавця. Це призводило до пікового споживання RAM у приблизно 200 % від розміру результуючого дерева.

Відповіддю на обмеження Wavefront став алгоритм Elastic Range (ER) (2011) [7]. Метод застосовує альтернативне двовимірне розбиття суфіксного дерева: вертикальне – для поділу на піддерева, і горизонтальне – для розбиття вхідних даних на незалежні блоки. Ключовою особливістю алгоритму є його еластичність. Розмір партицій динамічно коригується залежно від поточної форми дерева і розміру доступної пам'яті, що дозволяє ER ефективно працювати з рядками, що суттєво перевищують розмір RAM. Експериментальне дослідження, проведене на звичайному настільному комп'ютері, продемонструвало здатність алгоритму обробити увесь людський геном за 19 хвилин. Однак, головним недоліком ER залишається погана масштабованість паралельної версії через накладні витрати на синхронізацію між потоками виконання.

Великий стрибок у масштабованості забезпечив алгоритм DGST, представлений у 2019 році. DGST реалізований на платформі Apache Spark, яка є стандартом де-факто для розподіленої обробки великих даних у кластері. Алгоритм реалізує повний цикл MapReduce, виконуючи більшість проміжних обчислень в оперативній пам'яті. Це надає фреймворку численні переваги над конкурентами у швидкості, простоті використання, модульності та масштабованості. Однак, вузькими місцями архітектури Spark залишаються висока вартість I/O звернень до зовнішніх накопичувачів і необхідність в серіалізації/десеріалізації даних [8, 9]. Такі операції є невід'ємною частиною робочого процесу і включають зчитування вхідних даних з HDFS, міжвузлову передачу під час Shuffle-фази та запис проміжних результатів на диск.

Попри виявлені обмеження Spark-архітектури, розподілена побудова суфіксних структур залишається активним напрямком досліджень. Флік і Алуру (2015) показали, що паралельна побудова суфіксних масивів і LCP-масивів у розподіленій пам'яті може бути реалізована засобами MPI (Message Passing Interface) без залучення Spark: їхній алгоритм опрацював людський геном менш ніж за 8 секунд на 1024 процесорних ядрах [10]. Проблему надмірного споживання пам'яті частково вирішили Фішер і Курпіч (2019), запропонувавши легковажний розподілений алгоритм побудови суфіксних масивів, що потребує вдвічі менше пам'яті порівняно з аналогом і дозволяє обробляти вдвічі більший обсяг вхідних даних на тій самій апаратній платформі [11]. У напрямку побудови суфіксних дерев Глибовець та Діденко (2023) запропонували алгоритм побудови узагальнених суфіксних дерев на розподілених паралельних платформах, оптимальний за часовою складністю та споживанням пам'яті, що підтримує індексування суфіксів для рядків довільної довжини з великими алфавітами [12]. Водночас дослідження

Хааґ (2025) підтверджує, що надлишкове споживання пам'яті залишається відкритою проблемою: для більшості розподілених реалізацій цей показник сягає 30–60-кратного розміру вхідних даних; пропонований авторами адаптований DCX-алгоритм дозволяє скоротити його до 14–26 разів [13]. Таким чином, загальна тенденція у галузі спрямована на зниження інфраструктурних вимог та зменшення накладних витрат на передачу даних – саме ті обмеження, які є ключовими для підходу, запропонованого в цій роботі.

Натомість, запропонована в цій роботі архітектура переносить кластерну організацію Apache Spark на Node.js стек. Архітектура цього середовища базується на однопотоковій моделі виконання (Single-threaded Event Loop) і, на відміну від Apache, має неблокуюче асинхронне I/O. Критично важливим в такій архітектурі є запобігання блокуванню основного потоку під час виконання обчислювально інтенсивних задач. Саме ця вимога визначає архітектурні рішення, описані в наступному розділі.

Адаптація алгоритму. Адаптація DGST до середовища Node.js вимагає вирішення двох ключових проблем: забезпечення паралельного виконання в однопотоковому середовищі та організації ефективного спільного доступу до даних між потоками без накладних витрат на серіалізацію [14]. Оскільки DGST передбачає складні математичні операції над великими масивами даних, використання стандартної однопотокової моделі Node.js є неефективним і призводить до повної зупинки системи.

Одним з рішень такої проблеми є використання воркерних потоків, що програмно реалізує модуль worker_threads [15]. Їх ключовою особливістю є функціонування в окремих потоках єдиного процесу операційної системи. Проте ізоляція контекстів у вигляді окремих примірників V8 isolate, власних куп пам'яті та збирачів сміття зумовлює значні системні витрати на підтримку високорівневої об'єктної моделі [16, 17]. Комунікація через messagePort накладає обмеження на спільний доступ до даних, оскільки за замовчуванням механізми передачі даних вимагають їх серіалізації або передачі прав володіння.

Для подолання цих викликів існує третій, найбільш ефективний механізм – спільна пам'ять, яку надає об'єкт SharedArrayBuffer (SAB) [18]. SAB є фундаментальним примітивом у мові JavaScript, який дозволяє резервувати фіксований блок двійкових даних у фізичній пам'яті. При передачі буфера між потоками дані не серіалізуються і не дублюються, реалізуючи притаманний SAB принцип нульового копіювання (Zero-Copy). Для роботи з байтовою послідовністю в якості представлення було обрано Uint8Array. Цей вибір зумовлюється достатністю представлення символів у кодуванні UTF-8 та зручністю в адресації цих окремих символів.

Подібно до Apache Spark, основний потік системи механічно нарізає SAB на рівномірні спліти, які потім розподіляються між воркерами для паралельної обробки. Як результат, кожен воркер отримує ізольовану ділянку пам'яті, яку він лише читає або модифікує локально. Жодні два воркери не пишуть ту саму ділянку SAB, виключаючи можливість «стану гонитви». Інакше кажучи, це нівелює необхідність імплементації Atomics для контролю синхронізації потоків.

Таким чином, між елементами оригінальної та запропонованої архітектур можна побудувати таке відображення:

- Керуючий вузол (Driver Node) → Основний потік;
- Виконавчий вузол (Executor Node) → Воркерний потік (worker_threads);
- HDFS → SAB;
- Фізичний блок даних в HDFS → Логічний спліт SAB.

Незмінним для обох архітектур залишається MapReduce потік виконання.

Така структурна трансформація дозволяє зберегти перевірену часом логіку масштабування алгоритму DGST, але фундаментально змінює фізичну модель доступу до даних. Заміна розподіленої файлової системи (HDFS) на область спільної пам'яті (SAB) дозволяє усунути накладні витрати на серіалізацію та мережеву передачу даних, що притаманні Spark. Завдяки цьому архітектура переходить від моделі з інтенсивним вводом-виводом (I/O-bound) до моделі з ефективним використанням пам'яті (memory-bound).

Процес обробки даних адаптовано до середовища спільної пам'яті через чотири алгоритмічні фази:

1. **Фаза ініціалізації та попередньої обробки**. Вхідні текстові масиви об'єднуються в єдиний рядок із використанням роздільників із приватної зони Unicode [19], що гарантує відсутність конфліктів із алфавітом UTF-8. Основний потік розбиває SAB на спліти, межі яких розширюються на «хвіст» довжиною $windowSize - 1$ байтів. Це гарантує, що жоден суфіксний префікс не буде обірваний на фізичній межі розподілу даних між воркерами;
2. **Фаза ітеративного розбиття на S-префікси.** Метою цього етапу є пошук такого набору S-префіксів довжиною ковзного вікна $windowSize$, щоб частота кожного унікального префікса не перевищувала встановлений ліміт пам'яті F_m. Це гарантує, що будь-яке піддерево, побудоване на основі такого префікса, вміститься в RAM окремого воркерного потоку:
   a. **Мар.** Виконавчі ноди паралельно сканують виділені спліти методом ковзного вікна – це базова техніка перебору підрядків рядка, що полягає у послідовному зсуві підрядка фіксованої довжини на одну позицію праворуч до кінця рядка. Для оптимізації обчислень кожен воркер будує локальне частотне префіксне дерево (Frequency Trie), що дозволяє групувати входження за один прохід;

b. **Shuffle і Reduce.** Для забезпечення детермінованості розподілу локальні результати групуються за хеш-кодами префіксів. Основний потік агрегує ці дані, формуючи глобальну картину розподілу частот. Якщо частота префікса перевищує ліміт F_m, він помічається як «проблемний», $windowSize$ інкрементується, і цикл повторюється лише для відповідних сегментів;

3. Фаза розподілу завдань. Ключовим завданням даного етапу є запобігання виникненню вузьких місць, що спричинені нерівномірним розподілом обчислювального навантаження між воркерами. Цю проблему вирішує комбінований підхід на основі алгоритмів Bin Packing [20] та Number Partitioning [17], що групують завдання за сумарною частотою префіксів;

4. Побудова піддерев. Побудова фрагментів розподіленого дерева здійснюється методом SALA (Suffix Array + LCP Array) [22]. Воркери оперують структурою LCP-Range [4], яка дозволяє виконувати лексикографічне сортування суфіксів не з нуля, а зі зміщенням на довжину вже відомого спільного префікса, що значно знижує навантаження на CPU.

Підсумовуючи, перехід до архітектури memory-bound дозволяє поєднати гнучкість високорівневої об'єктної моделі JavaScript із продуктивністю низькорівневого маніпулювання байтами. Описані алгоритмічні фази забезпечують ефективну паралелізацію побудови суфіксного дерева, мінімізуючи накладні витрати на синхронізацію потоків та копіювання даних.

Програмна реалізація. Програмна реалізація алгоритму DGST у середовищі Node.js базується на гібридній моделі управління даними, що поєднує низькорівневу оптимізацію доступу до тексту та високорівневу об’єктну логіку JavaScript. Ключовим рішенням є використання SharedArrayBuffer як єдиного незмінного джерела істини для вхідних даних, тоді як усі проміжні структури (частотні префіксні дерева, мапи префіксів та вузли дерева) реалізовані у вигляді динамічних об’єктів JavaScript.

Для розмежування вхідних текстових масивів застосовано символи з приватної зони Unicode (U+E000 та U+E001), що гарантує відсутність конфліктів із алфавітом UTF-8 та зберігає лексикографічний порядок при сортуванні. Ефективність використання RAM досягається за рахунок дескрипторного підходу: замість копіювання фрагментів тексту, ієрархічні структури оперують виключно числовими індексами у спільному буфері.

Основний потік виконує роль асинхронного оркестратора, що координує фази MapReduce без блокування основного циклу. Центральною функцією координації є processIteratively(), яка реалізує ітеративне збільшення ковзного вікна з умовою виходу на основі стану частотного дерева. На кожній ітерації виконується послідовність Map/Shuffle/Reduce, під час якої головний потік накопичує локальні результати в двох акумуляторах: finalPrefixes і targetPrefixes. Це дозволяє воркерним потокам на повторних ітераціях ігнорувати вже оброблені сегменти, суттєво заощаджуючи час CPU.

Взаємодія воркерних потоків і основного потоку побудована на базі спеціального об’єкту Promise [23], що дозволяє керувати станом асинхронної операції через resolve (успішне завершення) та reject (виникнення помилки). Для розподілу завдань на будь-яку фазу основний потік створює масив Promise для кожного завдання та використовує Promise.all() для очікування завершення всіх воркерних потоків. Це реалізує програмний бар’єр: основний потік не переходить до наступної фази, поки всі воркерні потоки не повернуть свої результати.

Для забезпечення ефективного управління ресурсами логіка керування життєвим циклом потоків інкапсульована в класі WorkerPool. Ключовими завданнями класу є одноразова ініціалізація пулу воркерних потоків, розподіл завдань між ними за моделлю циклічного програмування (round-robin) та повернення результатів в основний потік. Перевикористання воркерних потоків мінімізує накладні витрати операційної системи на створення нових процесів.

Якщо основний потік відповідає за глобальну координацію, то воркерні потоки виступають ізольованими обчислювальними одиницями, що реалізують безпосередню CPU-інтенсивну обробку даних. Програмна реалізація зосереджена у модулі worker-node.js, який використовує API worker_threads для комунікації з головним потоком.

Воркерні потоки працюють в подіє-орієнтованій моделі, реагуючи на повідомлення від основного потоку через порт зв’язку. Центральним елементом комунікації є обробник події message, який реалізує диспетчеризацію задач на основі поля phase: «divide» для Map-фази, «reduce» для Reduce-фази та «divide» для фази побудови піддерев. Така архітектура дозволяє воркерному потоку обробляти різні типи завдань без перезапуску потоку, мінімізуючи накладні витрати на ініціалізацію.

Одним із ключових викликів при роботі з SAB у середовищі Node.js є необхідність постійного перетворення бінарних даних у нативні JavaScript-рядки для коректної роботи текстових алгоритмів. Для уникнення надлишких обчислень та неефективного використання пам’яті реалізовано механізм кешування декодованого вмісту буфера. Воркерний потік зберігає посилання на останній оброблений буфер; при отриманні нового завдання система перевіряє ідентичність посилання, і у разі збігу – повертає вже існуючий рядок.

Завдяки модульній структурі, воркерний потік функціонує як спеціалізований сервіс, що адаптує логіку обробки під конкретну фазу MapReduce:

1. **Map.** Воркерний потік сканує виділений спліт методом ковзного вікна для побудови локального частотного дерева. Використання фільтра targetPrefixes дозволяє ігнорувати вже оброблені сегменти, що прискорює повторні ітерації;
2. **Reduce.** На основі отриманої партиції воркерний потік агрегує частоти однакових префіксів у структуру Map і повертає її основному потоку у вигляді згорнутого масиву;
3. **Побудова піддерев.** Використовуючи кешований текст, воркерний потік знаходить входження префіксів за алгоритмом Кнута–Морріса–Пратта (KMP) [24] та формує структуру піддерева через масиви SA та LCP за один лінійний прохід.

Реалізована програмна модель доводить ефективність поєднання буферів масивів спільного використання (SAB) із архітектурою воркерних потоків (worker_threads) для побудови систем, що критично залежать від пропускної здатності оперативної пам'яті. Завдяки дескрипторному підходу та механізмам кешування декодованого тексту вдалося нівелювати накладні витрати на серіалізацію та повторне декодування UTF-8. Асинхронна координація через програмні бар'єри Promise.all() забезпечує стабільну паралелізацію фаз MapReduce.

Результати. Для оцінки продуктивності розробленої системи було проведено серію експериментальних запусків. Тестування виконувалося на обчислювальній машині з наступними технічними характеристиками: процесор AMD Ryzen 5 7460HS, 12 логічних ядер, 16 ГБ оперативної пам'яті, під управлінням ОС Windows 11. Всі вимірювання проводилися у середовищі виконання Node.js v24.12.0.

Важливою особливістю конфігурації експерименту стало використання 12 воркерних потоків. Такий вибір зумовлений архітектурою процесора: призначення одного потока на кожне логічне ядро дозволяє досягти максимального використання обчислювальних потужностей без накладних витрат на контекстне перемикання потоків (context switching).

Об'єктом дослідження виступив текстовий масив обсягом 2 млн рядків, який був розділений на чотири вибірки по 0,5 млн (~7 МБ), 1,0 млн (~13 МБ), 1,5 млн (~20 МБ) та 2,0 млн рядків (~25 МБ). Методика тестування передбачала проведення 12 контрольних запусків для кожного набору даних. Критеріями оцінки ефективності виступали: середній час обробки, пікове споживання RAM (MemoryLimit), кількість виявлених префіксів (SPrefixes) та кількість ітерацій головного циклу обробки, яка відображає кількість кроків алгоритму до досягнення фінального результату. Результати замірів зафіксовано у таблиці 1.

Таблиця 1

Результати замірів продуктивності системи

Кількість рядків	Середній час (с)	SPrefixes	MemoryLimit	Кількість ітерацій
0,5 млн	8,396586	608	144361	2
1,0 млн	15,406581	736	282613	2
1,5 млн	19,380307	788	418141	3
2,0 млн	25,929663	849	526693	3

Аналіз результатів демонструє пряму залежність між обсягом даних та споживанням ресурсів. При збільшенні кількості рядків у чотири рази (з 0,5 млн до 2,0 млн) показник MemoryLimit зріс лише у ~3,6 раза (з 144 361 до 526 693 одиниць). Оскільки приріст пам'яті відстає від приросту обсягу даних (коли даних стало в 4 рази більше, показник пам'яті зріс лише в 3,64 раза), це об'єктивно підтверджує ефективність буферів масивів спільного використання (SAB): 12 воркерів працюють з єдиним спільним масивом у пам'яті, не створюючи власних копій даних, що зазвичай призвело б до набагато стрімкішого зростання показників.

Найбільш вагомим доказом ефективності кешування є порівняння результатів для 1,0 млн та 1,5 млн рядків. Попри те, що обсяг даних зріс на 50 % і одночасно збільшилася кількість ітерацій головного циклу з 2 до 3, загальний час виконання збільшився лише на 4 секунди (з 15,4 с до 19,4 с). Якби воркери щоразу заново декодували один і той самий буфер для кожної нової ітерації, час обробки зростав би набагато швидше. Той факт, що додаткова ітерація майже не сповільнила систему, доводить: воркери використовують уже готовий декодований рядок із кешу.

Продуктивність системи підтверджується і підсумковою швидкістю на максимальному обсязі у 2 млн рядків, яка становить 77 132 рядки за секунду (2 000 000 рядків / 25,9 с). Навіть при зростанні кількості знайдених префіксів до 849 одиниць, що збільшує навантаження на етапі агрегації результатів, система зберігає високу продуктивність.

Висновки. У ході проведеного дослідження було реалізовано та досліджено ефективність алгоритму побудови суфіксних дерев DGST у середовищі Node.js із застосуванням підходу обробки даних в оперативній пам'яті для виконання ресурсоємних обчислень. Було розроблено багатопотоковий програмний модуль, у якому обчислювальні завдання асинхронно розподіляються між воркерними потоками в межах одного процесу операційної системи.

Експериментальне дослідження підтвердило працездатність системи на реальних текстових масивах. Зокрема, було встановлено, що час обробки зростає приблизно лінійно відносно розміру датасету, тоді як приріст пам'яті помітно відстає від приросту обсягу даних. Така закономірність свідчить про те, що застосовані оптимізації у вигляді буферів масивів спільного використання (SharedArrayBuffer), кешування декодованого буфера у воркерах та одноразової ініціалізації пулу із повторним використанням ефективно стримують зростання навантаження на RAM. Це є непрямим підтвердженням високої масштабованості системи в умовах збільшення вхідного датасету.

Водночас практичний досвід показав, що розроблена архітектура належить до моделі, яка критично обмежена обсягом оперативної пам'яті – фактором, що напряму визначає верхню межу продуктивності. Це накладає певні вимоги до середовища розгортання і є природним обмеженням in-memory підходу порівняно з повноцінним розподіленим кластером. Проте для задач, де кластерна інфраструктура є надлишковою або недоступною, така модель демонструє виправданий баланс між продуктивністю та простотою розгортання.

Перспективним напрямом подальших досліджень є повний відхід від високорівневої об'єктної моделі на користь бінарних структур даних для представлення проміжних результатів, зокрема використання TypedArrays для збереження вузлів суфіксного дерева та масивів індексів. Це дозволить уникнути надлишкового споживання ресурсів на рівні об'єктних структур та відкриє можливості для масштабування системи до обробки датасетів обсягом у десятки мільйонів рядків.

References:

1. Weiner, P. (1973), «Linear pattern matching algorithms», *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT)*, Iowa City, USA, pp. 1–11.
2. McCreight, E.M. (1976), «A space-economical suffix tree construction algorithm», *Journal of the ACM*, Vol. 23, No. (2), pp. 262–272.
3. Ukkonen, E. (1995), «On-line construction of suffix trees», *Algorithmica*, Vol. 14 (3), pp. 249–260.
4. Zhu, G., Guo, C., Lu, L. et al. (2019), «DGST: Efficient and scalable suffix tree construction on distributed data-parallel platforms», *Parallel Computing*, pp. 87–102.
5. GitHub (2026), *sab-mapreduce repository*, [Online], available at: https://github.com/allyxandraaa/sab-mapreduce
6. Ghoting, A. and Makarychev, K. (2009), «Serial and parallel methods for I/O efficient suffix tree construction», *Proceedings of the ACM SIGMOD International Conference on Management of Data*, *Providence*, USA, pp. 827–840.
7. Mansour, E., Allam, A., Skiadopoulos, S. and Kalnis, P. (2011), «ERA: efficient serial and parallel suffix tree construction for very long strings», *Proceedings of the VLDB Endowment*, Vol. 5, No. 1, pp. 49–60.
8. *HDFS Architecture Guide*, Apache Hadoop, [Online], available at: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
9. *Tuning Spark 3.5.0 Documentation*, Apache Spark, [Online], available at: https://spark.apache.org/docs/latest/tuning.html
10. Flick, P. and Aluru, S. (2015), «Parallel distributed memory construction of suffix and longest common prefix arrays», *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15), Austin, TX, USA*, 15–20 November, ACM, New York, pp. 16:1–16:10.
11. Fischer, J. and Kurpicz, F. (2019), «Lightweight distributed suffix array construction», *Proceedings of the 21st Workshop on Algorithm Engineering and Experiments (ALENEX 2019)*, San Diego, CA, USA, 7 January, SIAM, Philadelphia, pp. 27–38.
12. Hlybovets, A. and Didenko, V. (2023), «Constructing generalized suffix trees on distributed parallel platforms», *Cybernetics and Systems Analysis*, Vol. 59, Issue 1, pp. 49–60.
13. Haag, M., Kurpicz, F., Sanders, P. and Schimek, M. (2025), «Fast and lightweight distributed suffix array construction», *33rd Annual European Symposium on Algorithms (ESA 2025)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 351, pp. 47:1–47:18.
14. Casciaro, M. and Mammino, L. (2016), *Node.js Design Patterns*, 2nd ed., Packt Publishing Ltd, Birmingham, pp. 24–31.
15. *Worker threads*, Node.js, [Online], available at: https://nodejs.org/api/worker_threads.html
16. Sheludko, I. and Solanes, S.A. (2020), «Pointer Compression in V8», *V8 Blog*, [Online], available at: https://v8.dev/blog/pointer-compression
17. Bruni, C. (2017), «Fast properties in V8», *V8 Blog*, [Online], available at: https://v8.dev/blog/fast-properties
18. «SharedArrayBuffer», *MDN Web Docs*, [Online], available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

19. Microsoft (2024), *Private use area (PUA) characters and End-user-defined characters (EUDCs)*, [Online], available at: https://learn.microsoft.com/en-us/globalization/encoding/pua
20. Johnson, D.S. (1973), *Near-optimal bin packing algorithms*, 401 p.
21. Mertens, S. (2003), «The easiest hard problem: Number partitioning», *Computational Complexity and Statistical Physics*.
22. Manber, U. and Myers, G. (1993), «Suffix Arrays: a new method for on-line string searches», *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948.
23. Keller, A., «Discover Promises in Node.js», *Node.js*, [Online], available at: https://nodejs.org/en/learn/asynchronous-work/discover-promises-in-nodejs
24. Knuth, D.E., Morris Jr., J.H. and Pratt, V.R. (1977), «Fast pattern matching in strings», *SIAM Journal on Computing*, Vol. 6, pp. 323–350.

Зважій Дмитро Володимирович – аспірант, старший викладач Національного університету «Києво-Могилянська академія».

https://orcid.org/0000-0003-1705-3590.

Наукові інтереси:

- алгоритми та структури даних;
- системи управління базами даних.

Малій Олександра Михайлівна – студентка бакалаврської програми «Інженерія програмного забезпечення» Національного університету «Києво-Могилянська академія».

Наукові інтереси:

- веборієнтовані системи;
- алгоритми та структури даних.

Zvazhii D.V., Malii O.M.

Construction of distributed suffix trees

The paper addresses the problem of parallel suffix tree construction over large text arrays without the use of cluster infrastructure. The evolution of relevant algorithms is analysed – from the single-threaded methods of Weiner, McCreight, and Ukkonen to the distributed approaches of Wavefront, ER, and DGST. A key limitation of the DGST algorithm, implemented on the Apache Spark platform, is identified: significant overhead costs associated with data serialisation and I/O access to the distributed HDFS file system.

An architecture for adapting the DGST algorithm to the Node.js environment is proposed, based on the worker_threads module and the SharedArrayBuffer mechanism. The structural transformation maps Spark components onto Node.js counterparts: the driver node to the main thread, the executor node to the worker thread, and HDFS to SharedArrayBuffer. By applying the Zero-Copy principle and caching decoded buffer contents, the serialisation overhead inherent to Spark is eliminated, and the architecture shifts from an I/O-bound to a memory-bound model. Four algorithmic phases of the adapted method are described: initialisation and pre-processing, iterative partitioning into S-prefixes, the Map phase with construction of local frequency prefix trees, and Shuffle/Reduce with load balancing based on the Bin Packing and Number Partitioning algorithms.

An experimental evaluation on a text array of up to 2 million strings confirmed approximately linear growth of processing time relative to dataset size, and sublinear growth of memory consumption: when the data volume increased fourfold, the MemoryLimit indicator grew by only 3.6 times. The system throughput at maximum volume amounted to 77,132 strings per second.

Keywords: suffix tree; DGST; Node.js; worker_threads; SharedArrayBuffer; parallel computing; MapReduce; in-memory processing.

Стаття надійшла до редакції 30.12.2025.