УДК 004.42

А.М. ГЛИБОВЕЦЬ
Національний університет «Києво-Могилянська академія», Київ, Україна,
e-mail: *a.glybovets@ukma.edu.ua*.

В.О. ДІДЕНКО
Національний університет «Києво-Могилянська академія», Київ, Україна,
e-mail: *verochka1998@gmail.com*.

ПОБУДОВА УЗАГАЛЬНЕНИХ СУФІКСНИХ ДЕРЕВ НА РОЗПОДІЛЕНИХ ПАРАЛЕЛЬНИХ ПЛАТФОРМАХ

Анотація. Запропоновано алгоритм побудови узагальнених суфіксних дерев з використанням розподілених паралельних платформ, який є оптимальним з погляду як часової складності, так і використання пам'яті. Розподілений підхід до побудови дає змогу працювати з великими алфавітами та дуже довгими рядками. Алгоритм є ефективним щодо масштабованості на розподілених паралельних платформах і підтримує суфікси індексування для різноманітних довгих рядків, починаючи від одного довгого рядка до кількох довгих рядків різної довжини.

Ключові слова: суфіксне дерево, узагальнене суфіксне дерево, ERa, алгоритм, паралельна побудова суфіксного дерева.

ВСТУП

Суфіксне дерево [1, 2] є фундаментальною структурою даних для роботи із символьною інформацією і використовується під час розв'язування багатьох задач оптимізації оброблення тексту великого розміру. Суфіксне дерево (СД) зберігає рядок шляхом індексації всіх можливих суфіксів цього рядка та застосовується у різноманітних системах для забезпечення швидких операцій над рядками (зіставлення фраз, пошук найдовшого повторюваного підрядка у послідовності символів або виявлення максимальної кількості повторюваних фраз у довгій послідовності символів).

З огляду на інтенсивне зростання обсягів даних, які потребують машинного оброблення, дослідженню алгоритмів побудови оптимального СД приділяють велику увагу [3–6]. Значних зусиль докладено до побудови паралельних алгоритмів СД на основі MPI [3]. Однак, вони мають обмеження з погляду масштабованості та відмовостійкості у тому разі, коли здійснюється введення великих за розміром даних. Водночас постійно зростає попит на ефективні алгоритми для побудови суфіксних дерев на розподілених паралельних платформах, як-от Hadoop [7] і Apache Spark [8].

У цій роботі представлено ефективний і високомасштабований алгоритм побудови узагальнених дерев суфіксів на розподілених паралельних платформах. Він складається з двох основних етапів: поділу паралельного піддерева та побудови паралельного піддерева.

На першому етапі реалізовано нову стратегію розповсюдження даних. Далі запропоновано ефективний алгоритм поділу піддерева, який будує LCP-масив [9–12] у паралельний спосіб, тобто паралельно обчислює, скільки разів зустрічається підрядок. Щоб покращити балансування навантаження та зменшити витрати на читання даних, передбачено ефективну стратегію розподілу завдань між паралельними процесами.

На другому етапі під час побудови паралельного піддерева використано структуру даних LCP-Range та багатосторонній алгоритм сортування LCP-Merge [9–12] для паралельної побудови LCP-масиву. Також модифіковано

© А.М. Глибовець, В.О. Діденко, 2023

відомий алгоритм побудови дерев суфіксів Elastic Range (ERa) [1]. Метою цієї модифікації є покращення його можливостей оброблення рядків великої довжини, яка перевищує доступну пам'ять, і адаптація алгоритму для виконання на платформі Apache Spark [8].

1. СУФІКСНЕ ДЕРЕВО

Суфіксним деревом (СД) T для рядка S довжини m називають орієнтоване дерево, що має m листів, пронумерованих від 1 до m, в якому кожна вершина, окрім кореня, має щонайменше два відгалуження; кожне ребро марковане непорожнім підрядком із рядка S; жодна вершина не може мати ребра, маркування якого починається з однакової літери [1].

Важливою особливістю СД є те, що саме конкатенація маркувань ребер на шляху від кореня до листа дає суфікс рядка S, починаючи з i-ї літери.

Суфіксне дерево існує не для кожного рядка. Якщо суфікс рядка збігається з його префіксом, то шлях для першого суфікса не може бути завершений у листі (у графі з'явиться цикл). Для усунення такої ситуації під час аналізу рядка до алфавіту символів рядка додають ще одну літеру, якої в ньому немає. Зазвичай, цю літеру позначають \$.

Уточнимо загальне означення. Позначимо набір символів Σ, які діятимуть як алфавіт. Нехай $S = s_0, s_1, \ldots, s_{n-1}, \$$, де $s_i \in \Sigma$, $i \in [0, n-1]$, $\$ \notin \Sigma$. Тоді суфікс S, який позначають S_i, є послідовністю $s_i \ldots s_{n-1}\$$, $i \in [0, n-1]$. Аналогічно, префікс суфікса S_i є послідовністю $s_i \ldots s_j$, $j \in [i, n-1]$. Далі префікс суфікса S_i будемо називати S-префіксом. Зауважимо, що унікальність \$ гарантує те, що жоден суфікс не є префіксом іншого суфікса. Отже, СД T є деревом, яке індексує всі суфікси рядка S.

Послідовність символів (можливо, порожню) з алфавіту позначимо буквами r, s і t, де $|t|$ — довжина рядка t. Префікс w рядка t — це такий рядок, що $wv = t$ для деякого (можливо, порожнього) рядка v. Суфікс w рядка t — це такий рядок, що $vw = t$ для деякого (можливо, порожнього) рядка v.

Доведено, що компактне СД можна представити у вигляді, який потребує $O(n)$ пам'яті [1]. На рис. 1 наведено структуру даних СД для рядка S = BANANA [1] .

Подібно до СД, узагальнене суфіксне дерево (УСД) — це дерево, яке індексує всі суфікси для набору рядків. Суфікси кожного рядка представлені в узагальненому дереві суфіксів. Типовий підхід до побудови УСД для набору рядків полягає у додаванні унікального термінального символа (UTS) до кожного рядка, подальшому об'єднанні всіх рядків і побудові СД для об'єднаного рядка. У цьому разі введений допоміжний термінальний символ, який додають у кожному рядку D, використовують для того, щоб переконатися, що жоден суфікс не є підрядком іншого суфікса. У цьому документі всі рядки об'єднано безпосередньо без додавання термінального символа до кожного рядка. Натомість у цій реалізації використано інформацію про розташу-

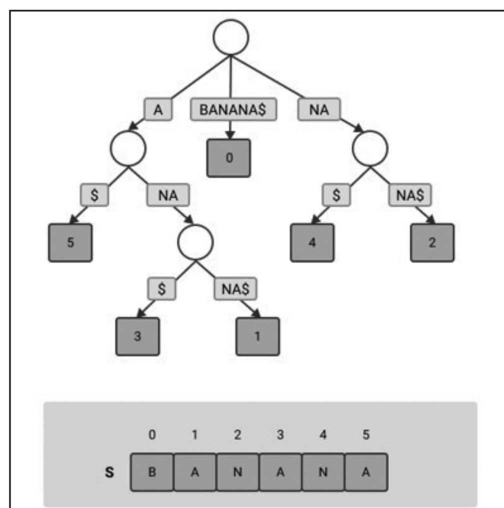

Рис. 1. Структура даних СД для рядка S = BANANA

вання кожного рядка в об'єднаному рядку, точніше, початкову та кінцеву позицію кожного рядка в об'єднаному рядку.

Приклад УСД для набору рядків $D = \{\text{ABAB}, \text{BABA}\}$ наведено на рис. 2. Листя дерева містять значення, яке складається з двох частин: ідентифікатора рядка та початкової позиції суфікса. Враховуючи, що СД є окремим випадком узагальненого дерева суфіксів, де розмір набору рядків D дорівнює одиниці (тобто зберігається один рядок), можна дійти висновку, що алгоритм побудови СД також застосовується до узагальненої версії, що дає змогу зосередитися на побудові СД для одного довгого рядка [2].

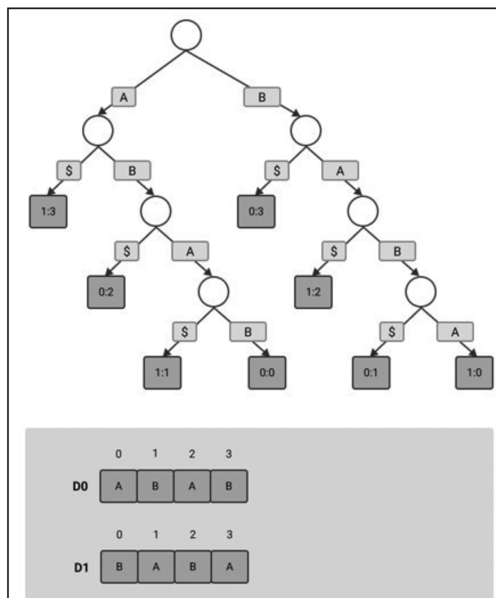

Рис. 2. Приклад узагальненої структури даних СД

Структура даних УСД зберігає не один рядок, а кілька — вона індексує набір суфіксів кожного рядка. Отже, узагальнене суфіксне дерево (УСД) забезпечує більшу гнучкість для прискорення різноманітних операцій під час роботи з текстом. Оскільки структура даних узагальненого СД є ефективною для оброблення рядків, її широко застосовують у багатьох програмних продуктах для прискорення різних завдань оброблення тексту, як-от: стиснення даних, кластеризації документів тощо.

Типовими завданнями, що виникають під час аналізу даних, особливо великих даних, є пошук фрази або фільтрація. У цих завданнях, якщо повний набір даних (набір рядкових даних) проіндексовано в УСД, усі рядки в наборі даних, які починаються з шуканої фрази, можна визначити за час $O(|\,searchedPhrase|)$. Якби СД не використовувалося, цей час становив би $O(|\,entireStringDataCollection| + |\,searchedPhrase|)$. Так само, значного прискорення можна було б досягти для інших рядкових операцій, як-от: зіставлення фрази, пошуку найдовшого підрядка, наявного в обох рядках, а також пошуку усіх загальних спільних підрядків у колекції рядкових даних [2–5].

З поширенням великих даних зростає попит на ефективні алгоритми побудови СД для великих алфавітів [5, 6]. Однак, побудова СД є складним процесом, що передбачає використання проміжних даних, які займають значну частину доступної пам'яті [7].

Щоб пришвидшити процес побудови СД і уможливити доступ до даних поза межами доступної пам'яті машини, створено кілька паралельних алгоритмів. Прикладами таких алгоритмів є WaveFront [2, 5, 6], PCF [2] і ERa [2]. Однак, слід взяти до уваги, що і WaveFront, і ERa дублюють усю колекцію рядкових даних для кожного обчислювального вузла, водночас кожен вузол обробляє призначені йому завдання побудови піддерева. Очевидно, що це створює проблему масштабованості у разі введення великих даних.

Якщо СД можна зберігати у пам'яті, то вже є багато швидких й елегантних рішень для побудови самого СД, яке справлятиметься із завданням в $O(|\,S\,|)$, де $|\,S\,|$ — довжина рядка. Прикладом такого алгоритму є алгоритм Укконена [8],

який зберігає високу ефективність доти, доки дані є меншими за розмір оперативної пам'яті. Далі його ефективність починає знижуватися. До того ж, побудоване дерево насправді є у багато разів більшим, ніж початкова колекція рядкових даних. Побудоване СД може збільшити обсяг використання пам'яті більше ніж у 30 разів [7].

Нині розподілені паралельні обчислювальні інфраструктури, як-от Hadoop [9] і Spark [10], стали стандартами для великомасштабного оброблення даних. Вони мають такі важливі характеристики, як висока масштабованість, простота використання та достатня відмовостійкість. На жаль, більшість сучасних алгоритмів паралельної побудови СД на основі MPI навряд чи можна інтегрувати із зазначеними платформами. Були спроби створити розподілене СД на цих платформах, однак вони накладають однакові обмеження щодо масштабованості, оскільки збір даних рядків дублюється для кожного обчислювального вузла (як у WaveFront і ERa). Крім того, ці підходи не надто ефективні для оброблення дуже довгих рядків. Тому в цій роботі створено новий алгоритм на основі модифікації алгоритму ERa для побудови УСД на розподілених паралельних платформах, який зберігає ефективність і водночас забезпечує велику масштабованість. Змодельований алгоритм підтримує суфікси індексування для різноманітних довгих рядків, від одного довгого рядка до кількох довгих рядків різної довжини.

Щоб виконати розподілене оброблення, усі рядки колекції рядкових даних об'єднано в один дуже довгий рядок (вхідний рядок), який потім поділено на розділи. Кожен розділ призначено обчислювальному вузлу та оброблено далі шляхом паралельного поділу піддерева та паралельної побудови піддерева. Тому суть створеної модифікації алгоритму ERa є такою: розроблено нову стратегію спільного використання даних для поділу піддерева та побудови піддерева у парадигмі паралелізму даних; уведено ефективний алгоритм поділу піддерева, що ґрунтується на паралельному методі обчислення частоти появи підрядка; для покращення балансу навантаження та зменшення витрат на читання запропоновано ефективну стратегію розподілу завдань побудови піддерева; на етапі побудови піддерева використано нову структуру даних LCP-Range [11] та універсальний алгоритм сортування LCP-Merge [11] для побудови паралельного масиву LCP [11]. Реалізацію прототипу випробувано на платформі Spark. Виконано експеримент з оцінювання ефективності, проведено її порівняння з оригінальною реалізацією алгоритму ERa.

2. МОДИФІКАЦІЯ АЛГОРИТМУ ERA ПОБУДОВИ РОЗПОДІЛЕНОГО УСД

Сучасні двоетапні алгоритми побудови СД, як-от WaveFront і ERa, не забезпечують достатньої масштабованості у разі введення дуже довгих рядків. Спочатку, на етапі поділу піддерева, підрахунок частоти для S-префіксів потрібно здійснювати, починаючи з вхідного рядка, кілька разів поспіль. Далі, під час фази побудови піддерева WaveFront і ERa дублюють вхідний рядок у кожному обчислювальному вузлі. Тому можна передбачити, що зі збільшенням розміру вхідного рядка ці алгоритми втратять ефективність. У цій роботі представлено новий алгоритм на основі модифікації ERa, який дає змогу розв'язати проблему масштабованості. Він містить ефективні розподілені паралельні алгоритми як для етапів поділу піддерева, так і для етапів побудови піддерева.

Розглянемо докладно новий алгоритм. Детально пояснимо кожен етап алгоритму побудови СД, який включає три основні частини, а саме розподіл даних,

паралельний поділ піддерева та паралельну побудову піддерева. Далі визначимо складність алгоритму і насамкінець наведемо результати експериментального тестування ефективності алгоритму та зробимо висновки.

2.1. Розподіл даних. Для поділу та створення піддерева у парадигмі паралельних даних спочатку розроблено нову стратегію розподілу даних, детально описану в цьому підрозділі.

Спершу, щоб забезпечити баланс між обчислювальними вузлами, вхідний рядок ділять на рівні частини і кожен розділ призначають обчислювальному вузлу. Оскільки розділи обробляються паралельно, кожному розділу, тобто кожній розділеній частині, призначають додатковий хвіст (за винятком останнього розділу). Це дає змогу забезпечити підрахунок частоти паралельного S-префікса для поділу піддерева, а також паралельне зіставлення S-префікса для підтвердження того, що побудови піддерева виконано правильно [12].

Ключовою проблемою під час розподілу даних є правильне визначення довжини хвоста для кожного розділу. Частотний поріг, позначений FM, визначають за допомогою підходу, подібного до того, що використовується в алгоритмі ERa. Цей поріг частоти не повинен перевищувати частоти S-префікса з набору [2]. Далі потрібно гарантувати, що паралельна частота обчислюється правильно і що S-префікс зберігається в паралельній парадигмі без помилок. Очевидно, що для цього довжина хвоста повинна бути принаймні такою самою, як максимальна довжина S-префіксів у наборі, який далі названо *maxPrefixLen*. У середньому значення *maxPrefixLen* можна обчислити на основі того, що початковий рядок отримано з випадкового симетричного розподілу Бернуллі. У цьому сценарії порогове значення частоти визначають так:

$$\frac{totalSizeOfInputString}{alphabetSize} \times maxPrefixLen,$$

де *totalSizeOfInputString* — загальна кількість символів в вхідній стрічці, *maxPrefixLen* — максимальна довжина S-префіксів у наборі, *alphabetSize* — розмір алфавіту. Тоді *maxPrefixLen* можна представити такою формулою:

$$\log_{alphabetSize} \frac{totalSizeOfInputString}{FM}.$$

Важливим є те, що кожному розділу призначають додатковий хвіст однакової довжини під час побудови СД. Однак, у разі створення узагальненого СД, довжина додаткового хвоста у різних розділах може бути різною. Ця різниця виникає через те, що за задумом УСД може зберігати більше одного рядка, бо кожен розділ, створений під час кроку розподілу даних, потенційно може мати кілька вхідних рядків. Беручи до уваги останнє, довжину хвоста для вхідного розділу можна обчислити у такий спосіб: $\min(len(RC), TAIL_LEN)$, де RC — вміст останнього рядка, що залишився у вхідному розділі, а $TAIL_LEN$ — довжина хвоста за замовчуванням.

2.2. Паралельний поділ піддерева. Після етапу підготовки даних, на якому вхідний рядок було поділено на частини (розділи), створено розподілене СД. Процес створення остаточного СД складається з двох основних етапів: поділу піддерева та побудови піддерева. Розглянемо ключові аспекти етапу поділу піддерева.

Етап поділу піддерева ґрунтується на паралельному підрахунку частот. На початку кожен обчислювальний вузол незалежно отримує доступ до розділу та створює локальну частотну спробу. Далі отримують відповідний S-префікс p разом із його частотою fp для кожного кінцевого вузла локальної частоти *trie*.

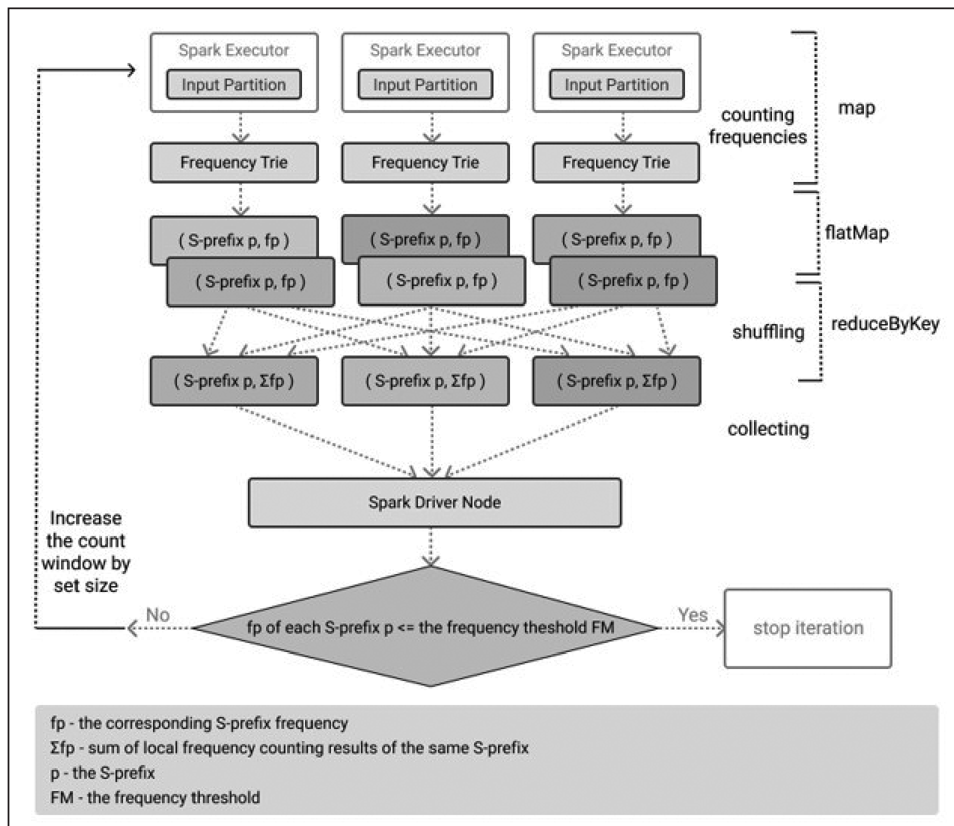

Рис. 3. Схема етапу паралельного поділу на піддерево

На наступному кроці здійснюється перемішування всіх зібраних пар S-префікса та відповідної частоти (p, fp) так, що один і той самий обчислювальний вузол отримує результати обчислення локальної частоти того самого S-префікса. Коли обчислення локальної частоти того самого S-префікса надсилаються до обчислювального вузла, вони незалежно агрегуються та передаються до вузла-драйвера, який збирає результати та вирішує, чи завершено процес поділу піддерева. Якщо це так, то його можна зупинити. Описаний процес, що виконується на платформі Spark, наведено на рис. 3.

Обчислення частоти появи S-префікса є основним кроком у поділі піддерева. Оскільки вхідний рядок розбивають на частини (під час виконання етапу розподілу даних, частоти можна обчислювати в кожній частині паралельно. Крім того, зазначений додатковий хвіст, який призначають кожному розділу, забезпечує гарантію того, що обчислення паралельної частоти виконано правильно.

До того ж, для ефективного обчислення частот S-префіксів реалізовано структуру даних frequency trie [11]. Частота *trie* — це деревоподібна структура даних, яка зберігає частоту S-префікса, обчислену із заданого вхідного рядка S. Далі її визначають у такий спосіб. Спершу край частоти *trie* та вузол дерева частот позначають e і v відповідно; label(e) вказує на символ, який зустрічається в S; path(v) вказує конкатенацію реберних значень на шляху від кореня до вузла v; label(v) вказує частоту, з якою path(v) зустрічається в рядку S. Отже, можна побачити, що S-префікс p у частоті *trie* представлений path(v), а fp насправді є міткою (v) у частоті *trie*. За визначенням label(v) можна вважати сумарними мітками всіх дочірніх вузлів, якщо вузол v є внутрішнім. У такий самий спосіб, використовуючи інфор-

мацію про шлях і мітку всіх листових вузлів, можна відновити частоту *trie*. Структуру даних частоти *trie* наведено на рис. 4.

Враховуючи цю частотну характеристику *trie*, можна зменшити загальну кількість ітерацій (тобто доступів уведення/виведення) шляхом обчислення частот S-префіксів різної довжини простим скануванням вхідного рядка.

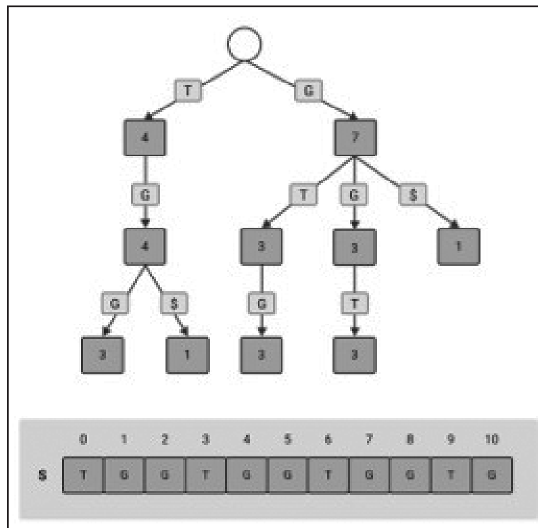

Рис. 4. Структура даних частоти *trie*

Останньою застосованою технікою для поділу піддерева є додатковий підрахунок вікон, наведений на рис. 3. Загальний процес поділу піддерева у паралельний спосіб організовано у кілька ітерацій, під час яких вікно підрахунку збільшують на визначений розмір кроку. Позначимо *winit* початкове вікно підрахунку. Тоді у межах першої ітерації підраховують частоти всіх S-префіксів довжини *winit*. Після завершення першої ітерації S-префікси додають або до набору P, або до набору R залежно від того, чи підходить частота S-префікса до порогового значення FM, чи ні. Вузол драйвера перевіряє, чи містить набір R S-префікси з частотами, вищими за поріг FM. Якщо це так, набір R транслюється кожному виконавцю для подальшого оброблення. Якщо R порожній, весь процес завершується, а ітерація припиняється. На наступній ітерації розмір вікна збільшують на визначений розмір кроку, позначений *wstepSize*. Тепер позначимо *wnext* вікно збільшеної кількості. Отже, частоти всіх S-префіксів, які мають довжину, що вписується в інтервал [*winit*, *wnext*], обчислюються в цій ітерації. Як зазначено раніше, процес ітерації припиняється, коли набір R більше не містить S-префіксів з частотами, що перевищують поріг FM [12].

На останньому кроці створюють частотну спробу з використанням S-префіксів у результівному наборі P, а надлишкові вузли видаляють шляхом обрізання всіх вузлів, дочірніх для вузла у частотній спробі, яка має мітку, що не перевищує поріг FM.

2.3. Паралельна побудова піддерева. Після етапу поділу піддерева, описаного в попередньому підрозділі, виконують етап побудови піддерева, який складається з таких кроків: розподіл завдань для побудови піддерева, локальне сортування суфіксів, сортування LCP-Merge та побудова піддерева. Схему процесу паралельної побудови піддерева зображено на рис. 5.

Спочатку для паралельної побудови піддерева здійснено розподіл відповідних завдань між обчислювальними вузлами. Далі створено кожне піддерево за допомогою нового багатогранного алгоритму на основі LCP-Merge.

Слід вказати основні фактори, які впливають на ефективність етапу розподілу завдань, і пояснити, як ми розглядали ці фактори, щоб покращити загальну продуктивність кінцевого етапу паралельної побудови піддерева. Здебільшого на ефективність розподілу завдань побудови піддерева впливають два основні фактори: кількість викликів уведення/виведення та підтримання балансу робочих

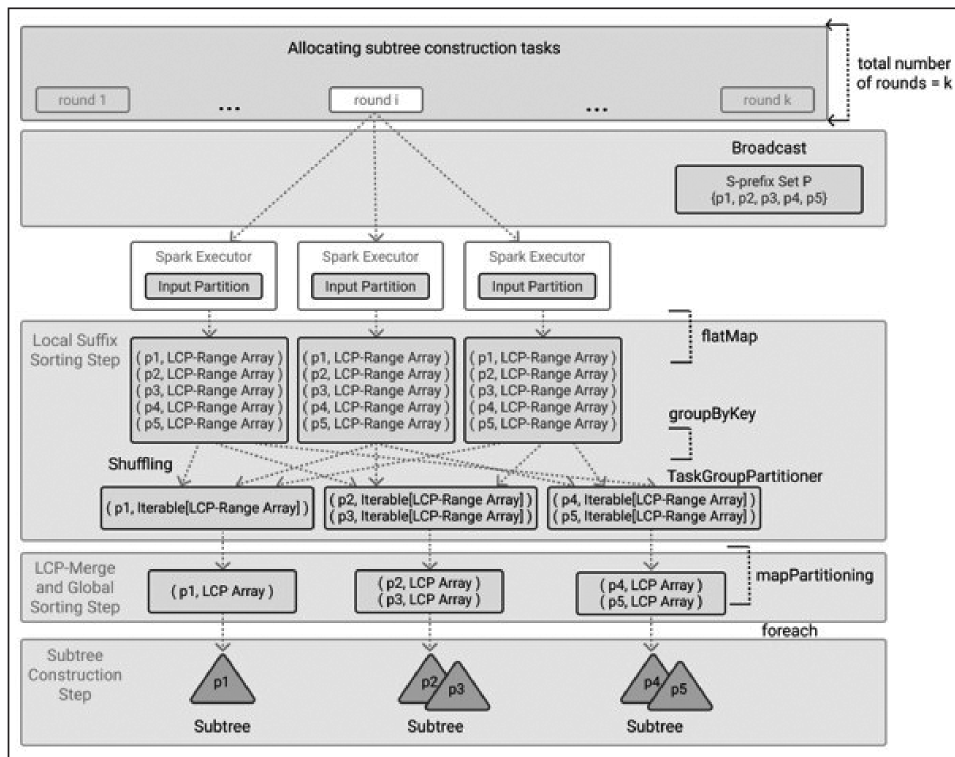

Рис. 5. Схема паралельної побудови піддерева

вузлів. Вплив першого фактора можна пояснити тим, що кожне завдання має отримати доступ до всього вхідного рядка під час побудови піддерева. Якщо буде надто багато маленьких піддерев для оброблення, то витрати на читання будуть високими. Що стосується другого фактора, то піддерева можуть відрізнятися за розміром. Якщо певні обчислювальні вузли перевантажуються, а інші вузли залишаються неактивними, це вплине на загальну продуктивність. Тому слід розробити стратегію розподілу завдань зі збалансованим навантаженням.

Для розв'язання обох зазначених проблем запропоновано нову стратегію розподілу завдань для побудови піддерева. Її суть полягає у використанні комбінації двох алгоритмів, а саме Bin-Packing [13] і Number-Partitioning [14, 15] .

Першу проблему (зниження вартості читання структури піддерева) можна розв'язати шляхом групування якомога більшої кількості піддерев подібно до ERa, що зменшить загальну кількість операцій читання. Оскільки кожна сформована група піддерев має загальну суму S-префіксів, яка не перевищує порогової частоти FM, усі піддерева в групі можна побудувати в одній обчислювальній задачі. Мінімізацію кількості груп піддерева можна розглядати як задача пакування контейнера. Тут можна застосувати алгоритм Bin-Packing, де Bin є групою S-префікса, а ємність Bin відповідає порогу FM. Алгоритм Bin-Packing розв'язує задачу знаходження мінімальної кількості груп, які можна сформувати з урахуванням того, що сума частот у кожній групі не перевищує порогу FM.

Водночас алгоритм Bin-Packing не може гарантувати, що для кожної групи навантаження буде збалансованим. Задачу балансу навантаження можна розглядати як задачу розподілу чисел. Число відповідає частоті S-префікса. Отже, застосовуючи алгоритм Number-Partitioning (алгоритм ділення чисел) можна переконатися в тому, що суми частот у кожній групі будуть майже однаковими. Однак, алгоритм розподілу чисел повинен заздалегідь знати загальну кількість

груп. Отже, алгоритм Bin-Packing і алгоритм Number-Partitioning використано в комбінації на етапі остаточної паралельної реалізації піддерева.

Перш ніж перейти до деталей реалізації паралельної побудови піддерева, потрібно визначити загальну кількість раундів побудови піддерева. На практиці ступінь паралелізму p можна розглядати як кількість виконавців на платформі Spark. Оскільки вхідний рядок розбито на k розділів, кількість груп завдань побудови піддерева дорівнює добутку k та p. Отже, можна дійти висновку, що розподіл усіх завдань на побудову піддерева складається з k раундів.

У кожному раунді всі виконавці Spark обробляють p груп завдань побудови піддерева паралельно. На початку кожного раунду всі задіяні S-префікси транслюються до кожного Spark Executor (робочого вузла). Далі, щоб створити локальний масив LCP-Range для кожного S-префікса, суфікси локально сортуються на кожному виконавцеві у такий спосіб. Кожному виконавцю Spark надається вхідний розділ (частина розділеного вхідного рядка); використовуючи вхідний розділ, кожен виконавець починає зіставлення всіх S-префіксів у відповідному вхідному розділі; наприклад, для S-префікса p, усі суфікси, які починаються з p у кожному вхідному розділі, сортуються. Далі на основі відсортованих суфіксів будується локальний масив LCP-Range.

Після етапу локального сортування суфіксів і побудови масиву LCP-Range масиви одного S-префікса групуються разом. Водночас S-префікси, які є частиною однієї групи завдань побудови піддерева, розташовані в одному розділі. Отже, усі піддерева в одній групі можна побудувати під час виконання одного обчислювального завдання. Під час етапу сортування LCP-Merge усі локальні масиви LCP-Range з однаковим S-префіксом об'єднують методом порівняння з урахуванням LCP-Range. У результаті отримують впорядкований масив LCP.

Загальний процес побудови містить такі основні етапи: локальне сортування суфіксів, сортування LCP-Merge і сортування за невпорядкованими інтервалами. Структуру даних LCP-Range створено для кожного суфікса у масиві відсортованих суфіксів.

Структуру даних LCP-Range наведено на рис. 6. Як видно, вона зберігає інформацію для порівняння двох рядків і представлена елементом, що складається з трьох частин: першого символа меншого рядка, де обидва рядки починають відрізнятися; послідовності символів більшого рядка в діапазоні від індексу, де обидва рядки починають відрізнятися до заданої довжини діапазону; числа зсуву — індексу, де обидва рядки починають відрізнятися. Визначений триплет позначено так: $lcp_range(s_1, s_2) = (c,$ діапазон, зсув).

Для кращого розуміння розглянемо приклад, наведений на рис. 7 (структура даних LCP-Range для рядка S1 = ”abcdeef” і рядка S2 = ”abcdffj”). Припустимо, що є два рядки S1 і S2, які мають вигляд послідовності символів: S1 = “abcdeef” і S2 = “abcdffj”. З-поміж двох рядків S1 менший за S2. Далі визначимо найдовший спільний префікс (LCP) для S1 і S2. Ним є LCP = “abcd”. Це означає, що зміщення, за якого два порівнювані рядки (S1 і S2) починають відрізнятися, буде індексом першого символа після LCP, який насправді є довжиною LCP. Тому довжина LCP є довжиною abcd і становить чотири символи. Отже, зміщення у наведеному прикладі дорівнює 4. Символ за індексом зсуву 4 у меншому рядку S1 — це “e”. Нарешті, довжина діапазону є константою, визначеною заздалегідь. Нехай у цьому прикладі довжина діапазону дорівнює 2; тоді послідовність символів у більшому рядку S2, починаючи з індексу зсуву 4, має довжину два символи і є такою: “ff”. Отже, структуру даних LCP-Range для S1 і S2 обчислюють як такий триплет: $lcp_range(S1, S2) = (e, ff, 4)$. Тут менший ря-

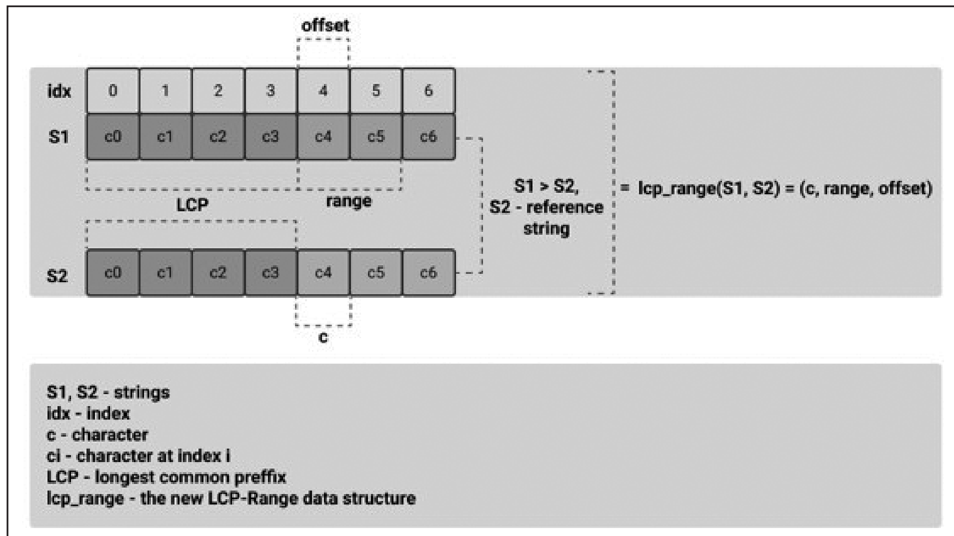

Рис. 6. Структура даних LCP-Range

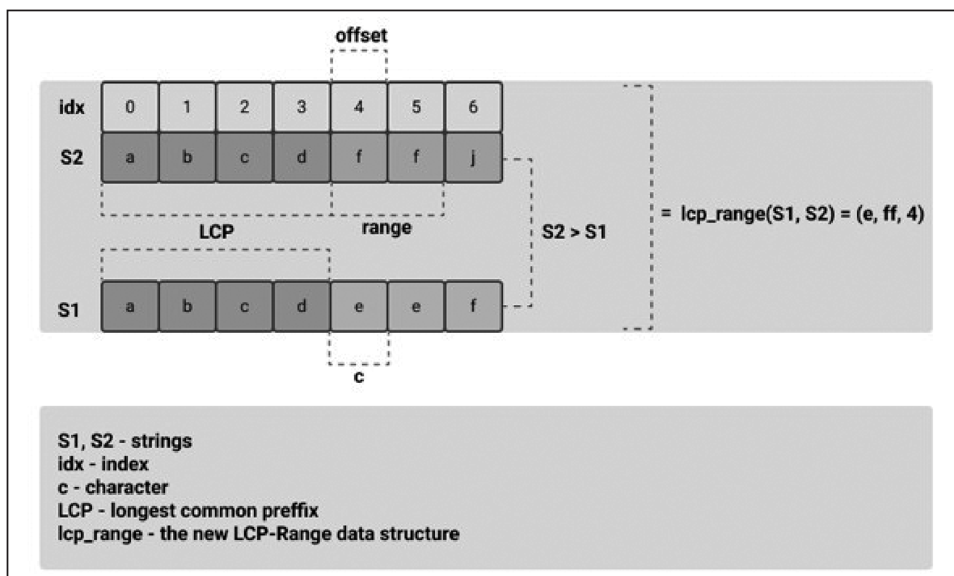

Рис. 7. Структура даних LCP-Range для рядка S1=abcdeef і рядка S2=abcdffj

док із двох називають також еталонним рядком. Отже, у наведеному прикладі еталонним рядком є S1, оскільки S1 менший за S2.

Зауважимо, що масив LCP-Range зберігає структури даних LCP-Range, точніше, інформацію про LCP-Range пар суфіксів, які постійно йдуть одна за одною в лексикографічно відсортованому масиві суфіксів. Наприклад, для послідовності суфіксів суфікс1, суфікс2, суфікс3, суфікс4, суфікс5 інформація діапазону LCP буде розрахована для кожної з таких пар: <суфікс1, суфікс2>, <суфікс2, суфікс3>, <суфікс3, суфікс4>, <суфікс4, суфікс5>. Тому результівний масив LCP-Range складатиметься з таких структур даних LCP-Range: lcp_range(суфікс1, суфікс2), lcp_range(суфікс 2, суфікс 3), lcp_range(суфікс 3, суфікс 4), lcp_range(суфікс 4, суфікс 5).

Оскільки зменшувальний суфікс (найменший суфікс) не має посилального рядка, перший елемент масиву LCP-Range замінено початковим діапазоном символів найменшого суфікса. Далі обчислено LCP між послідовними суфіксами. Як зазначено раніше, всі масиви LCP з однаковим S-префіксом, які були згенеровані після етапу сортування локальних суфіксів, об'єднано. Результатом є формування глобально впорядкованого масиву LCP. Процес об'єднання реалізовано на основі LCP-Merge з використанням дерева сортування, а саме дерева програшів, яке є різновидом дерева турнірів [11].

Слід зазначити, що дерево програшів має чудову властивість: для кожного гравця з одного шляху є фіксований прямий шлях, що рухається в напрямку кореневого вузла [11]. Припустимо, що остаточним переможцем турніру є шлях w, тоді наступний гравець p, що входить до w, потрапляє до трійки переможених. Гравець p спочатку порівнюється з інформацією про діапазон LCP, що зберігається в його батьківському вузлі. Два діапазони LCP мають однаковий контрольний рядок, який є остаточним переможцем останнього турніру. Після порівняння діапазонів LCP переможець переходить до наступного батьківського вузла.

Інформація про переможця та діапазон LCP, які зберігаються в наступному батьківському вузлі, досі мають той самий еталонний рядок, тому порівняння діапазону LCP можна продовжувати, доки не буде визначено остаточного переможця, тому решта процесу є подібною: кожні два порівнювані LCP діапазони містять один і той самий посилальний рядок [16]. Насамкінець будують суфіксне піддерево.

3. РОЗРАХУНОК СКЛАДНОСТІ НОВОГО АЛГОРИТМУ

Оскільки змодельований алгоритм складається з двох основних етапів (етапу паралельного поділу піддерева та етапу побудови паралельного піддерева, тобто паралельної побудови масиву LCP), розраховано складність кожного етапу.

Складність етапу паралельного поділу піддерева можна обчислити на основі максимальної довжини S-префіксів із частотами, нижчими за поріг частоти FM, складності побудови частотної спроби для кожного вхідного розділу та загальної кількості ітерацій, які відбулися раніше, а також за умови, що не було більше S-префіксів з частотами, що перевищують поріг частоти FM. Нехай *maxPrefixLen* позначає максимальну довжину S-префіксів із частотами, нижчими за поріг частоти FM. У середньому *maxPrefixLen* займає пам'ять:

$$O\left(\log_{alphabetSize}\frac{totalSizeOfInputString}{FM}\right),$$

а в гіршому випадку *maxPrefixLen* займає щонайбільше: $O(totalSizeOfInputString)$. Далі, враховуючи, що вхідний рядок S поділений на p розділів, складність побудови частоти *trie* для кожного вхідного розділу можна обчислити так:

$$O\left(\frac{totalSizeOfInputString}{p}\times w\right),$$

де p — кількість розділених розділів; а w — вікно підрахунку для кожної ітерації під час кроку поділу піддерева. Крім того, останній фактор, а саме загальну кількість ітерацій, можна обчислити у такий спосіб:

$$O\left(\frac{maxPrefixLen}{wstepSize}\right),$$

де *wstepSize* — розмір кроку вікна підрахунку. Отже, після об'єднання обчислень кожного фактора загальна складність етапу поділу паралельного піддерева є такою:

$$O\left(\frac{totalSizeOfInputString}{p} \times \log_{alphaberSize}\left(\frac{totalSizeOfInputString}{FM}\right) \times p\right).$$

Однак, на практиці виявляється, що *maxPrefixLen* є дуже малим. Він не є суттєвим, його можна не брати до уваги, тому загальна складність фактично дорівнює $O(totalSizeOfInputString)$.

Складність етапу побудови паралельного піддерева об'єднує в собі складність етапу локального сортування та складність етапу багатостороннього сортування LCP-Merge. Складність етапу локального сортування залежить від загальної кількості суфіксів, які мають однаковий початковий S-префікс. Ця загальна кількість у найгіршому випадку становитиме $O(totalSizeOfInputString)$. Оскільки піддерева попередньо розділені, загальна кількість зменшується і може щонайбільше дорівнювати порогу частоти FM. Складність кроку локального сортування для S-префікса можна обчислити так:

$$O\left(\frac{FM}{p} \times \log_2 \frac{FM}{p} \times p\right),$$

де p — кількість розділених розділів. Нарешті, складність етапу багатостороннього сортування LCP-Merge можна обчислити як:

$$O(FM \times \log_2 k),$$

де k — загальна кількість раундів на етапі побудови паралельного піддерева.

4. АНАЛІЗ РЕЗУЛЬТАТІВ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ

У реалізації цієї роботи використано Apache Hadoop як віртуальну розподілену файлову систему. На момент написання статті автори проводили експерименти з налаштуванням локальної машини, що працює під керуванням операційної системи Ubuntu 20.04 (версії Apache Spark 1.6.3 і Apache Hadoop 2.6.5). Напрямком майбутніх досліджень є подальше створення фізичного кластера та проведення тестів на більших обсягах даних задля визначення областей, які слід додатково оптимізувати зі збільшенням навантаження.

Щоб порівняти приріст продуктивності реалізованого алгоритму з оригінальним найсучаснішим алгоритмом ERa, останній також реалізовано на тій самій машині і одному середовищі. Крім того, і розробленому алгоритму, і оригінальній реалізації ERa було надано один і той самий вхідний набір даних. Використано відкриті загальнодоступні набори даних, взяті з платформи Kaggle [17], розміщені в розподіленій файловій системі Hadoop і надані обом реалізаціям алгоритму, що порівнюються. Протягом усього процесу ми виводили статистику виконання на консоль і зберігали у вихідних файлах.

Графіки задежності часу роботи від розміру вхідного рядка для запропонованого та оригінального алгоритму ERa наведено на рис. 8. За результатами порівняння статистики часу виконання можна дійти висновку, що новий підхід для розподілу узагальненої конструкції СД є ефективним і дійсно забезпечив приріст продуктивності приблизно у 2–2.5 рази відносно продуктивності алгоритму ERa.

Отже, можна стверджувати, що методи оптимізації, застосовані в алгоритмі, представленому у цій роботі для процесу побудови СД на розподілених платформах, справді покращили загальну продуктивність. Основною причиною

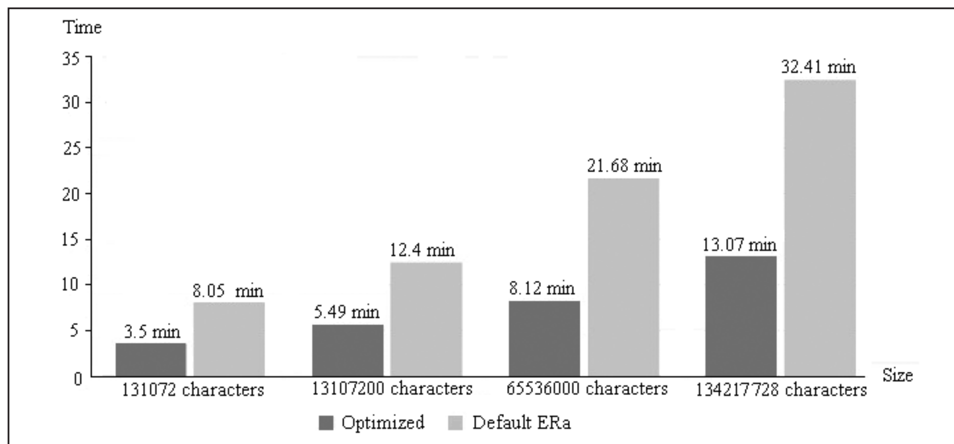

Рис. 8. Діаграми залежності часу роботи (у хвилинах) від розміру вхідного рядка (у символах) для оптимізованого та оригінального алгоритмів ERa

збільшення продуктивності є те, що алгоритм підрахунку частоти, який використовується в ERa для поділу піддерева, є послідовним, а пропонований є паралельним. До того ж в ERa вікно підрахунку подовжується лише на одиницю, що призводить до багатьох ітерацій і високих витрат на введення-виведення. З іншого боку, у запропонованому алгоритмі використано метод паралельного підрахунку частоти для прискорення етапу поділу піддерева. Крім того, розмір кроку вікна підрахунку збільшується на більше число, тому кількість ітерацій зменшується. Як наслідок, запропонований алгоритм перевершує ERa на етапі поділу піддерева та забезпечує кращі результати у загальних вимірюваннях продуктивності. Напрямком майбутніх досліджень є подальше вдосконалення запропонованого алгоритму та досягнення більшого приросту продуктивності за рахунок розподіленого зберігання в пам'яті.

ВИСНОВКИ

Метою роботи було покращення алгоритму побудови СД під час роботи з великими алфавітами та дуже довгими рядками, які перевищують доступний обсяг пам'яті. У результаті розроблено новий алгоритм для побудови високомасштабованого УСД на розподілених паралельних платформах. Весь процес розбито на три основні етапи: етап підготовки даних, де вхідний рядок був розділений, а розділи розподілені між обчислювальними вузлами; етап поділу піддерева; етап побудови піддерева.

Для поділу піддерева та створення піддерева в парадигмі паралельних даних застосовано стратегію спільного використання даних. Далі запропоновано ефективний алгоритм поділу піддерева на основі паралельного підрахунку частот. Потім розглянуто ефективну стратегію для розподілу завдань побудови піддерева, яка може зменшити витрати на читання та досягти балансу навантаження на окремих вузлах. Насамкінець, на етапі побудови піддерева використано структуру даних LCP-Range й алгоритм сортування LCP-Merge для створення масиву LCP паралельно.

Експериментальні результати, отримані на Apache Spark, свідчать про те, що запропонований алгоритм значно перевершує найсучасніший алгоритм ERa, забезпечуючи прискорення приблизно у 2–2.5 рази. Отже, отриманий алгоритм для побудови розподіленого УСД є цілком підхожим для програмних систем, у яких є потреба у покращенні операції з великими текстовими даними та які ґрунтуються на розподіленій системі. Крім того, запропонований алгоритм може ефективно побудувати УСД для набору рядків великого розміру.

СПИСОК ЛІТЕРАТУРИ

1. Weiner P. Linear pattern matching algorithms. *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory (SWAT 1973)* (15–17 October 1973, Iowa City, Iowa, USA). Iowa City, 1973. P. 1–11.
2. Mansour E., Allam A., Skiadopoulos S., Kalnis P. ERA: Efficient serial and parallel suffix tree construction for very long strings. *Proc. VLDB Endowment (PVLDB).* 2011. Vol. 5, N 1. P. 49–60.
3. Kleppmann M. Designing data-intensive applications. O'Reilly Media, Inc., 2017. 614 p.
4. Comin M., Farreras M. Efficient parallel construction of suffix trees for genomes larger than main memory. *Proc. 20th European MPI Users' Group Meeting (EuroMPI'13)* (15–18 September 2013, Madrid, Spain). Madrid, 2013. P. 211–216.
5. Gobonamang T.K., Mpoeleng D. Counter based suffix tree for DNA pattern repeats. *Theoretical Computer Science*. 2020. Vol. 814, Iss. C. P. 1–12.
6. Ghoting A., Makarychev K. Suffix trees. In: Encyclopedia of Parallel Computing. Padua D. (Ed.), Boston, MA: Springer, 2011. P. 1945–1955. https://doi.org/10.1007/978-0-387-09766-4_464.
7. Ibarra O.H., Zhang L (Eds.). Computing and combinatorics. *Proc. 8th Annual International Conference, COCOON 2002* (15–17 August 2002, Singapore). Singapore, 2002. 614 p.
8. Ukkonen E. On-line construction of suffix trees. *Algorithmica*. 1995. Vol. 14. P. 249–260.
9. HDFS architecture guide. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
10. Apache spark. URL: https://spark.apache.org/.
11. Eberle A. Parallel multiway LCP-mergesort. Bachelor Thesis. 2014. 61 p.
12. Zhu G., Guo C., Lu L., Huang Z., Yuan C., Gu R., Huang Y. DGST: Efficient and scalable suffix tree construction on distributed data-parallel platforms. *Parallel Computing.* 2019. Vol. 87. P. 87–102.
13. The bin packing problem. URL: https://developers.google.com/optimization/bin/bin_packing.
14. Schreiber E.L., Korf R.E., Moffitt M.D. Optimal multi-way number partitioning. *Journal of the ACM.* 2018. Vol. 65, Iss. 4. P. 1–61.
15. Flick P., Aluru S. Parallel construction of suffix trees and the all-nearest-smaller-values problem. *Proc. 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (29 May – 2 June 2017, Orlando, Florida, USA). Orlando, 2017. P. 12–21.
16. Ng W., Kakehi K. Merging string sequences by longest common prefixes. *IPSJ Digital Courier*. 2008. Vol. 4. P. 69–78.
17. Datasets. URL: https://www.kaggle.com/datasets.

A. Hlybovets, V. Didenko

CONSTRUCTION OF GENERALIZED SUFFIX TREES ON DISTRIBUTED PARALLEL PLATFORMS

Abstract. This paper proposes an algorithm for constructing generalized suffix trees using distributed parallel platforms, which is optimal from the point of both time complexity and memory consumption. The distributed construction approach allows operating with large alphabets and very long strings. The algorithm is efficient for scalability on distributed parallel platforms. It supports indexing suffixes for various long strings, ranging from a single long string to multiple long strings of varying lengths.

Keywords: suffix tree, generalized suffix tree, ERa, algorithm, parallel construction of a suffix tree.

Надійшла до редакції 08.09.2022