УДК: 004.658:004.65 https://doi.org/10.15407/pp2026.01.014

Д.В. Зважій , С.С. Гороховський

ВПРОВАДЖЕННЯ ІНДЕКСІВ У POSTGRESQL

У статті досліджено процес проєктування та реалізації нового індексного методу доступу в системі управління базами даних PostgreSQL на прикладі індексу на основі суфіксного дерева. Суфіксні дерева забезпечують оптимальну теоретичну складність пошуку, однак класичні описи цієї структури даних припускають необмежену пам'ять, довільні маніпуляції з вказівниками та однопотоковий доступ — умови, які PostgreSQL не забезпечує. У роботі послідовно розглянуто ключові архітектурні підсистеми PostgreSQL, що безпосередньо впливають на проєктування індексних структур: багатопроцесну модель «один процес на з'єднання», сторінкову організацію зберігання даних із фіксованим розміром блоку 8 КБ, а також дворівневу систему керування пам'яттю на основі спільного пулу буферів та ієрархічних контекстів пам'яті. На прикладі конкретної реалізації проаналізовано ключові проєктні рішення, з якими стикається розробник нового індексу: вибір стратегії відображення логічних вузлів дерева на фізичні сторінки, організацію внутрішньосторінкового зберігання ребер змінної довжини з розділенням на ідентифікаційну та змістову частини, проєктування спеціальної області сторінки для навігаційних метаданих і суфіксних посилань, а також механізми ідентифікації типів сторінок. Окрему увагу приділено відкритим проблемам реалізації — стисненню міток ребер, обмеженням паралелізму алгоритму Укконена та можливостям підтримки index-only scan. Показано, що впровадження нової індексної структури в системі управління базами даних рівня PostgreSQL є не лише задачею адаптації алгоритму до конкретного програмного інтерфейсу, а й узгодженням його з внутрішніми інваріантами, протоколами та логікою взаємодії підсистем, які визначають допустимий простір архітектурних рішень.
Ключові слова: PostgreSQL, індекси, методи доступу, структури даних, керування пам'яттю, узагальнені суфіксні дерева, архітектура баз даних

D. Zvazhii , S. Gorokhovsky

IMPLEMENTING INDEXES IN POSTGRESQL

This article explores the process of designing and implementing a new index access method in the PostgreSQL database management system, using a suffix tree–based index as a case study. Suffix trees provide optimal theoretical search complexity; however, classical descriptions of this data structure assume unlimited memory, arbitrary pointer manipulation, and single-threaded access — conditions that PostgreSQL does not provide. The paper systematically examines the key architectural subsystems of PostgreSQL that directly affect index design: the "one process per connection" multi-process model, page-based storage organization with a fixed 8 KB block size, and the two-level memory management system based on the shared buffer pool and hierarchical memory contexts. Using a concrete implementation as an example, the paper analyzes the key design decisions a developer faces when building a new index: the strategy for mapping logical tree nodes onto physical pages, the organization of intra-page storage for variable-length edges with separation into identification and payload components, the design of the page special area for navigational metadata and suffix links, and mechanisms for page type identification. Particular attention is given to open implementation challenges — edge label compression, parallelism limitations of Ukkonen's algorithm, and the feasibility of supporting index-only scans. It is shown that implementing a new index structure in a DBMS of PostgreSQL's caliber is not merely a matter of adapting an algorithm to a specific programming interface, but also of aligning it with the internal invariants, protocols, and subsystem interaction logic that define the permissible space of architectural decisions.
Keywords: PostgreSQL, indexes, data structures, memory management, generalized suffix trees, database - architecture

Вступ

У сфері систем управління базами даних (СУБД) ефективність доступу до даних є визначальним чинником, що впливає на масштабованість системи та рівень затримок. Із переходом обсягів даних від гігабайтного до петабайтного масштабу обчислювальні витрати, пов'язані з операціями доступу, трансформуються з другорядного аспекту на критичне місце архітектури. Базовим механізмом доступу

до даних у системі PostgreSQL є послідовне сканування (Sequential Scan), яке характеризується часовою складністю $O(n)$, де n — це потужність множини, або таблиці. Попри прийнятність лінійного підходу для малих обсягів даних, у середовищах із високою пропускною здатністю така складність на практиці може виявитися неефективною.

У цих умовах індексні структури стають важливим елементом архітектури СУБД. PostgreSQL вирізняється серед багатьох інших СУБД не лише тим, що має відкритий вихідний код, а також завдяки наявності високорозширюваної індексної інфраструктури. Окрім традиційних B-дерев [1], PostgreSQL низку альтернативних методів доступу, — зокрема, Hash [2], GiST [3], SP-GiST [4][5], GIN [6] і BRIN [7] — кожен з яких орієнтований на обробку специфічних типів даних і семантик запитів. Що більш суттєво, PostgreSQL надає розробникам можливість визначати власні методи доступу до індексів, відкриваючи простір для експериментування з новими структурами даних і адаптації підходів індексування до предметно-орієнтованих задач. Така розширюваність робить PostgreSQL не лише промисловою системою керування базами даних, а й ефективною платформою для проведення наукових досліджень. Попри високу гнучкість, розроблення нового типу індексу для PostgreSQL залишається складним і нетривіальним завданням. Реалізація такого індексу потребує глибокого розуміння внутрішнього програмного інтерфейсу PostgreSQL, моделі конкурентного доступу, механізмів керування буферами, журналювання попереднього запису (write-ahead logging, WAL), а також особливостей взаємодії з планувальником запитів. Рішення, що ухвалюються на рівні індексу, зокрема, щодо організації сторінок, стратегій розбиття або гарантування узгодженості, мають далекосяжні наслідки для коректності, продуктивності та підтримуваності всієї системи. Як наслідок, багато потенційно цінних індексних концепцій залишаються недостатньо дослідженими або реалізуються поза межами СУБД, зокрема, в прикладних пошукових рушіях чи зовнішніх системах зберігання даних.

У попередній статті [8] основну увагу було зосереджено на адаптації специфікації Access Methods API (програмного інтерфейсу методів доступу) [9], яку PostgreSQL надає для імплементації індексу на базі суфіксного дерева. Натомість у цій статті процес розроблення індексів для PostgreSQL розглядається як із концептуальної, так і з практичної перспективи. Замість зосередження винятково на використанні наявних типів індексів аналізується, яким чином у межах розширюваного програмного інтерфейсу методів доступу PostgreSQL (Access Methods API) можна проєктувати та реалізовувати нові індексні структури. Окрему увагу приділено аналізу ключових особливостей архітектури PostgreSQL [10] та їхнього впливу на проєктування індексів. Як практичний приклад розглянуто розробку індексу на базі суфіксного дерева [11].

Аналіз останніх досліджень та публікацій

Як уже було зазначено вище, PostgreSQL надає чітко визначений інтерфейс для створення власних методів доступу до індексів і постачається з шістьома вбудованими типами індексів. Розширюваність цієї системи спирається на дві фундаментальні концепції. Геллерстайн, Нотон і Пфеффер [12] запропонували узагальнене дерево пошуку (GiST), яке об’єднує високо збалансовані індекси, зокрема, B+-дерева та R-дерева, у межах єдиного розширюваного програмного інтерфейсу. Проте GiST не здатне репрезентувати за своєю природою незбалансовані структури, зокрема, префіксні дерева (trie) або суфіксні дерева. Валід і Ільяс [4] запропонували для цього SP-GiST — фреймворк для просторово-роздільних незбалансованих дерев (trie, k-d-дерева, квадродерева). Елтабах, Елтаррас і Ареф [13] реалізували SP-GiST у PostgreSQL і продемонстрували суттєве зростання продуктивності, зокрема, представивши реалізацію суфіксного дерева для пошуку підрядків.

Попри підтримку незбалансованих структур, SP-GiST має критичне обмеження: кожен запис листка відповідає рівно одному ключу та одному ідентифікатору кортежу (TID). Для суфіксних структур це створює проблему, оскільки один рядок s довжини l породжує l суфіксів, які мають посилатися на той самий TID (tuple identifier). Обмеження «один ключ — один TID» унеможливлює повноцінну емуляцію поведінки суфіксного дерева або суфіксного масиву в межах SP-GiST. Нещодавно Шьоманс, Ареф, Зіманьї та Сакр [14] запропонували MGiST і MSP-GiST — розширення GiST і SP-GiST із підтримкою множинних записів, що дозволяють декомпозувати один об'єкт бази даних на кілька індексних записів за допомогою користувацького методу ExtractValue. Хоча ці підходи розроблялися для індексування траєкторій і визнають проблему обмеження «один запис на кортеж», їх досі не застосовано до суфіксних структур.

Що ж до стуктур даних, які зберігають суфікси: суфіксні дерева (Вайнер [15], Маккрейт [16], Укконен [17]) і суфіксні масиви (Манбер і Майєрс [18]) - то вони давно є ґрунтовно дослідженими. Проте практичні реалізації цих структур у межах СУБД залишаються вкрай рідкісними. Сюй, Чен, Хуан, Ху та Нонг [19] запропонували SAES — систему, що замінює інвертований індекс Elasticsearch індексом на основі суфіксного масиву для повнотекстового пошуку в гетерогенних даних; це одна з небагатьох спроб інтегрувати суфіксне індексування в наявну пошукову інфраструктуру.

Таким чином, існує очевидна прогалина: попри значну кількість алгоритмічних розробок, лише небагато структур реалізовано як вбудовані методи доступу до індексів у СУБД, а суфіксні дерева та масиви практично відсутні в промислових рушіях баз даних. Запропонована робота спрямована на заповнення цієї прогалини шляхом реалізації суфіксного індексу як власного методу доступу PostgreSQL та документування проєктних рішень, зумовлених її архітектурними обмеженнями (фіксований розмір сторінки 8 КБ, менеджер буферів, вимоги WAL), які алгоритмічна література здебільшого залишає поза увагою.

Реалізація індексу на базі суфіксного дерева

Задача пошуку підрядків — знаходження всіх рядків, у яких текстовий стовпець містить заданий шаблон, є поширеною в базах даних. Стандартні B-деревні індекси ефективно прискорюють префіксні запити (наприклад, LIKE 'pattern%'), однак не допомагають у випадку ведучих байдужих символів (LIKE '%pattern%'), що змушує систему виконувати послідовне сканування колонки. Суфіксні дерева забезпечують пошук за шаблоном за час $O(m + k)$ [20], де m — довжина шаблону, незалежно від розміру тексту, а k - кількість можливих повторів шаблону. Це робить суфіксні дерева теоретично привабливими для цієї задачі. Втім, класичні описи суфіксних дерев у підручниках припускають необмежену пам'ять, довільні маніпуляції з вказівниками та однопотоковий доступ. PostgreSQL не надає жодної з цих передумов. Натомість він оперує сторінками фіксованого розміру, спільним пулом буферів із суворими протоколами pinning, обов'язковим журналюванням попереднього запису (WAL) і конкурентним доступом з боку кількох бекендів. Інтеграція суфіксного дерева в таке середовище вимагає ухвалення проєктних рішень, які теоретична література зазвичай не розглядає. Цей розділ зосереджений на деталях реалізації індексу на базі суфіксного дерева. Спочатку окреслено архітектурні обмеження PostgreSQL, після чого розглядаються важливі проєктні рішення з якими зіштовхується розробник під час реалізації. Для конкретизації наводяться приклади із конкретної реалізації індексу на базі суфіксного дерева [11].

PostgreSQL зберігає всі дані у сторінках фіксованого розміру — зазвичай 8192 байти (8 КБ), який задається на етапі компіляції параметром BLCKSZ. Це обмеження є всеосяжним: у його межах працюють кожна індексна сторінка,

сторінка heap і записи журналу WAL. Для суфіксного дерева це одразу породжує низку питань. Вузол із великою кількістю вихідних ребер може не вміститися в одну сторінку. Мітка ребра може мати довільну довжину. Розмір сторінки не підлягає зміні — реалізації мають проєктуватися з урахуванням цього обмеження. Фактично доступний для даних простір сторінки є меншим за BLCKSZ. Кожна сторінка містить 24-байтовий заголовок (PageHeaderData) і також зазвичай резервує певний простір наприкінці сторінки для типоспецифічних метаданих (так звана special area). Для сторінки розміром 8 КБ приблизно 8140 байтів доступні для безпосереднього вмісту, а у випадку використання значної спеціальної області — навіть менше. Кожна сторінка PostgreSQL має стандартну внутрішню структуру. Заголовок сторінки містить вказівники (pd_lower, pd_upper), які окреслюють межі між зайнятою та вільною частинами сторінки. Методи доступу можуть зарезервувати спеціальну область (special area) наприкінці сторінки для типоспецифічних метаданих.

Розмір спеціальної області фіксується під час ініціалізації сторінки. Саме тут методи доступу зазвичай зберігають навігаційні вказівники (на батьківські та сусідні сторінки), прапорці типу сторінки та ідентифікаційні маркери. Проєктування цієї структури є одним із ключових ранніх рішень, оскільки її розмір і вміст безпосередньо впливають як на доступний простір для даних, так і на можливості еволюції індексної структури.

Менеджер буферів PostgreSQL опосередковує весь доступ до сторінок. Сторінки ніколи не зчитуються безпосередньо з диска; натомість бекенд-процеси

Лістинг 1: Ініціалізація спеціальної частини сторінки

```
PageInit(page, BLCKSZ,
sizeof(MyOpaqueData));
opaque = (MyOpaque) PageGetSpe-
cialPointer(page);
```

запитують буфери за чітко визначеним протоколом:

- Читання: ReadBuffer(relation, blockno) — фіксує (pin) сторінку в пулі буферів.
- Блокування: LockBuffer(buffer, mode) — захоплює спільне або виключне блокування.
- Доступ: BufferGetPage(buffer) — повертає вказівник на сторінку.
- Модифікація: MarkBufferDirty(buffer) — сигналізує, що сторінку було змінено.
- Звільнення: UnlockReleaseBuffer(buffer) — знімає блокування.

Утримання буфера в стані pinned запобігає витісненню сторінки з пулу, а утримання блокування перешкоджає конкурентним модифікаціям або забезпечує стабільність читання. Критично важливим наслідком є унеможливлення збереження сирих вказівників на сторінках між операціями з буферами: після звільнення буфера будь-який вказівник на його вміст стає недійсним. Для алгоритмів обходу дерев, які концептуально “спускаються” шляхом від кореня до листка, це означає необхідність вибору між двома підходами: або одночасно утримувати кілька буферів у стані pinned (що підвищує ризик взаємних блокувань), або повторно зчитувати сторінки після їхнього звільнення. Обидва варіанти накладають суттєві обмеження на проєктування структур даних і алгоритмів навігації в межах індексів PostgreSQL.

Розподілювач пам’яті PostgreSQL (palloc) працює в межах контекстів пам’яті — ієрархічних арен, які можна скидати або знищувати масово. Код методів доступу зазвичай виконується в контекстах відповідного кортежу (per-tuple contexts), що скидаються після оброблення кожного кортежу.

Із цього випливає важливе правило: не можна зберігати вказівники на пам’ять, виділену через palloc, за межами оброб-

Лістинг 2: Приклад зміни контексту під час індексації кортежу

```
oldCtx = MemoryCon-
textSwitchTo(buildState->tmpMem-
Ctx);
MemoryContextSwitchTo(oldCtx);
MemoryContextReset(buildState-
>tmpMemCtx); PageGetSpe-
cialPointer(page);
```

лення окремого кортежу. Будь-які допоміжні структури даних мають або розміщуватися в контексті з довшим часом життя, або відновлюватися заново для кожного кортежу.

PostgreSQL гарантує надійність збереження даних за допомогою журналювання попереднього запису (write-ahead logging, WAL): усі зміни мають бути зафіксовані в журналі до того, як модифікована сторінка буде записана на диск. Для методів доступу це має два ключові прояви:

- Критичні секції. Модифікації, які повинні бути атомарними, обгортаються в START_CRIT_SECTION() / END_CRIT_SECTION(). Усередині критичної секції будь-яка помилка призводить до PANIC (перезапуску бази даних), а не до подовження неузгодженого стану системи.

- WAL-записи. Складні методи доступу генерують власні типи WAL-записів. Простіші реалізації можуть покладатися на повні зображення сторінок (full-page images) під час початкової побудови індексу, уникаючи детального журналювання кожної окремої зміни.

Для інкрементних вставок коректне журналювання WAL вимагає проєктування форматів записів і реалізації процедур повторного відтворення (redo), що є складним і трудомістким завданням.

PostgreSQL визначає методи доступу до індексів за допомогою структури

Лістинг 3: Ключові функції зворотнього виклику методу доступу

```
amroutine->ambuild =
stree_build;

amroutine->ambuildempty =
stree_empty;

amroutine->aminsert = stree_in-
sert;

amroutine->ambeginscan =
stree_beginscan;

amroutine->amgettuple =
stree_gettuple;

amroutine->amendscan =
stree_endscan;
```

IndexAmRoutine [8], яка задає підтримувані можливості та набір зворотних викликів (callbacks), показаний на лістингу 3. Інтерфейс також декларує можливості методу доступу: чи може індекс забезпечувати унікальність, чи підтримує багатостовпцеві ключі, чи здатен повертати кортежі у впорядкованому вигляді тощо. Для суфіксного дерева, орієнтованого на пошук підрядків, більшість із цих можливостей відсутня: індекс не забезпечує унікальність, не підтримує впорядкування результатів і працює лише з одним текстовим стовпцем.

Під час проєктування індексу виникає необхідність зберігання різних структур, тому індекс може містити сторінки різних типів — сторінки метаданих, вузли дерева, сторінки переповнення для великих записів, а також сторінки для відстеження вільного простору. Під час читання сторінки реалізація методу доступу повинна однозначно визначити її тип. Крім того, діагностичні інструменти на кшталт pg_filedump отримують істотну користь від можливості ідентифікувати тип індексної сторінки без знання прикладного контексту. Для розв’язання цієї задачі застосовуються різні підходи:

- фіксовані номери блоків — окремі номери блоків мають фіксоване призначення (наприклад, мета-сторінка в блоці 0);

- використання відведених констант, записаних у визначене місце сторінки;

- збереження міток типу сторінки у спеціальній області сторінки (special area).

Схема з фіксованими номерами блоків може спрацювати, якщо є можливість попередньо зарезервувати під них пам'ять, але непридатна для сторінок, які будуть виділені динамічно під час вставок. Числа-константи забезпечують надійну ідентифікацію, проте споживають простір і потребують координації, щоб уникнути колізій між різними методами доступу. У практиці PostgreSQL зазвичай поєднують компактні мітки типу сторінки з фіксованим ідентифікатором методу доступу, розміщеним у спеціальній області сторінки (наприклад, streePageId = 0xCDEE). Такий підхід дає змогу розрізняти типи сторінок як усередині індексу, так і між різними індексними структурами, забезпечуючи зручність під час налагодження та валідації сторінок під час обходу — за незначної додаткової вартості простору, оскільки спеціальна область зазвичай уже містить навігаційні метадані.

Якщо говорити про стуктури, які повинен зберігати індекс, то можна виділити такі складові дерева як вузли, ребра та ідентифікатори кортежів (TID - tuple identifier), які буде повертати індекс. Ці типи даних суттєво відрізняються за розмірами, шаблонами доступу та характером зростання, що створює нетривіальну задачу відображення їх на сторінки фіксованого розміру PostgreSQL так, щоб забезпечити ефективну побудову індексу та швидкий пошук. Розміри вузлів суфіксного дерева можуть істотно варіюватися. Вузол поблизу кореня може мати вихідні ребра для сотень різних символів, тоді як вузол глибоко в дереві часто має лише одне ребро. Це породжує питання відображення: як

Лістинг 4: Приклад використання міток сторінки

```
#define STREE_META
(1<<0)
#define STREE_ROOT
(1<<1)
#define STREE_INTERNAL_NODE
(1<<2)
#define STREE_DATA_PAGE
(1<<3)

typedef struct STreeNodePage-
OpaqueData {
    uint16 flags;
    /* ... інші поля ... */
} STreeNodePageOpaqueData;
```

співвіднести логічні вузли дерева для зберігання на фізичних сторінках. Можна виділити три основні підходи:

- Один вузол на сторінку: кожен вузол дерева займає рівно одну сторінку розміром 8 КБ, а номер блоку (BlockNumber) слугує його ідентифікатором — без потреби в додатковій адресації.

- Кілька вузлів на сторінці: дрібні вузли пакуються разом для зменшення втрат простору. Це вимагає внутрішньосторінкової адресації (номер блоку + зсув) і ускладнює логіку росту вузлів.

- Вузли, що розтягуються на кілька сторінок: великі вузли можуть мати сторінки-продовження, що дозволяє підтримувати довільні розміри, але додає складності під час обходу й синхронізації.

Модель "один вузол — одна сторінка" природно узгоджується з менеджером буферів PostgreSQL. Закріплення (pinning) вузла означає фіксацію рівно одного буфера, а його звільнення — звільнення того самого буфера. Відсутні часткові блокування вузла, міжсторінкова узгодженість і складна адресна арифметика. Основний недолік цього підходу — неефективне використання простору: листковий вузол з

Лістинг 5: Представлення ребра

```
typedef struct EdgeId {
    pg_wchar  firstChar;
    uint16    payloadOffset;
    uint16    payloadLength;
} EdgeId;

typedef struct EdgePayload {
    BlockNumber destination;
    uint16      labelLength;
    char        label[];
} EdgePayload;
```

одним ребром і мінімальними метаданими може займати лише близько 100 байтів із 8 КБ, що означає до 98% втрат. Для дерева з мільйонами дрібних вузлів це може становити значні накладні витрати. Водночас суфіксні дерева мають корисну властивість: кількість внутрішніх вузлів обмежена сумарною довжиною індексованих рядків. Дерево, що індексує N символів, містить не більше ніж N внутрішніх вузлів. Отже, просторові накладні витрати, хоч і відчутні, зростають лінійно — $O(n)$ сторінок — і не є неконтрольованими. З огляду на це, для початкової реалізації підхід "один вузол на сторінку" пропонує переконливий баланс між простотою та коректністю. У власній реалізації [11] був використаний розширений варіант, який дозволяє вузлу розростатися на нову сторінку, якщо було використано увесь вільний простір. Оптимізації на кшталт пакування дрібних вузлів доцільно відкласти до наступної ітерації — коли стане зрозуміло, що саме використання простору, а не складність алгоритмів, є головним вузьким місцем.

Вузол суфіксного дерева також зберігає набір вихідних ребер, кожне з яких складається з мітки змінної довжини та вказівника на цільовий вузол. У процесі побудови суфіксного дерева алгоритм Укконена [17] багаторазово виконує три базові операції над ребрами:

- пошук ребра, що починається з заданого символу;
- вставка нового ребра з певною міткою та призначенням;
- поділ наявного ребра, коли його мітка скорочується на позиції k та перенаправляється до нового внутрішнього вузла.

Дизайн сторінки індексу має ефективно підтримувати всі ці операції, оскільки кожна з них по-різному модифікує вміст сторінки, а обсяг змін безпосередньо впливає на розміри WAL-записів і частоту позначення буферів як "брудних". Найчастішою операцією є пошук: на кожному кроці оброблення вхідного рядка алгоритм перевіряє, чи існує ребро з відповідним початковим символом. Це вимагає організації ребер у формі, зручній для пошуку — наприклад, шляхом зберігання у відсортованому вигляді з можливістю бінарного пошуку або за допомогою хешування. Вставка додає нові дані до сторінки: якщо ребра зберігаються компактно й упорядковано, вставка потребує зсуву всіх наступних елементів. У схемах з апендиксним додаванням корисних даних і окремим індексом вставка самих даних має амортизовану складність $O(1)$, а підтримка впорядкування — $O(n)$ за кількістю ребер. Ключовою операцією, яка впливає на дизайн структури сторінки, є операція поділу ребра. Під час розщеплення ребра e на позиції k його мітка, наприклад, ABCDEF, перетворюється на префікс AB, створюється новий внутрішній вузол із ребром CDEF, а призначення початкового ребра змінюється на цей новий вузол. Вирішальним є питання: чи можна скоротити мітку на місці, не переміщуючи байти в пам'яті, адже переміщення даних істотно збільшує обсяг модифікацій сторінки та відповідне WAL-журналювання. Ефективним розв'язанням цієї проблеми є відокремлення структури ребра, необхідної для пошуку (наприклад, першого символу), від його корисного навантаження — повної мітки та вказівника призначення.

Це дозволяє організувати сторінку, яка зростатиме у два напрямки (Рис.1):

- масив ідентифікаторів ребер зростає від заголовка сторінки, зберігаючи впорядкованість за першим символом;

- корисні дані ребер додаються апендиксом з боку спеціальної області сторінки, тобто у протилежному напрямку.

За такої організації вставка нового ребра з початковим символом c виконується через бінарний пошук позиції, зсув елементів масиву ідентифікаторів, запис нового ідентифікатора та додавання корисних даних у верхній частині вільного простору.

Архітектура сторінки, що дозволяє рости в обидва напрямки, розв'язує задачу зберігання ребер, однак вузол суфіксного дерева містить значно більше, ніж лише ребра. Під час побудови та використання дерева алгоритм повинен переміщатися між вузлами різними шляхами:

- переходити за суфіксними посиланнями під час побудови за Укконеном,

- підніматися до батьківських вузлів у певних сценаріях відновлення або перевірки коректности,

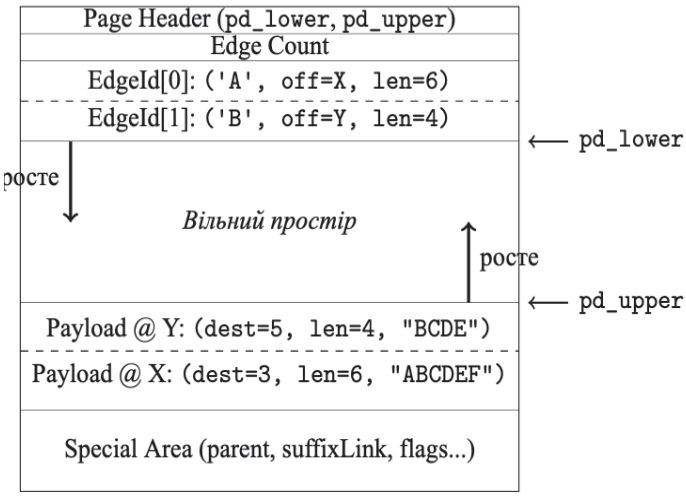

Рис. 1. Дизайн сторінки, яка зберігає ребра

- ітеруватися по сторінках даних під час збирання ідентифікаторів кортежів (TID).

PostgreSQL резервує наприкінці кожної сторінки простір для даних, специфічних для методу доступу, — до якого звертаються через PageGetSpecialPointer(). Для вузлів суфіксного дерева ця спеціальна область містить як навігаційні вказівники, так і ідентифікаційну інформацію про тип сторінки. Зокрема, тут зберігаються:

- **parentNode** - забезпечує можливість руху вгору по ієрархії. Хоча алгоритм Укконена здебільшого виконує обхід дерева від кореня до листків, у деяких ситуаціях, наприклад, під час відновлення або перевірки цілісності структури, потрібен підйом до батьківського вузла. Явне зберігання посилання на батька дозволяє уникнути підтримки окремого стеку обходу.

- **suffixLink** - ключове поле для досягнення лінійної складності O(n) під час побудови дерева за алгоритмом Укконена [17]. Після створення або відвідування внутрішнього вузла, що відповідає суфіксу $S[i..j]$, це посилання вказує на вузол для $S[i+1..j]$. Такий перехід між спорідненими суфіксами виконується за сталий час. Без механізму суфіксних посилань складність побудови зростала б до $O(n^2)$.

- **itemPointersStart** - вказує на першу сторінку в ланцюжку сторінок, що зберігають TID. Вузол, який відповідає поширеному підрядку, може відповідати тисячам рядків у heap; ідентифікатори кортежів не зберігаються у вузлі разом із ребрами, а виносяться в окремі пов'язані сторінки, щоб не перевантажувати основну структуру вузла.

- **prevSiblingNode** та **nextSiblingNode** - формують двобічно зв'язаний список сторінок переповнення для TID та ребер. Коли сторінка даних заповнюється, виділяється нова й приєднується до ланцюжка через ці вказівники.

- **streePageId** - число-константа (наприклад, 0xCDEE), що однозначно ідентифікує сторінку як сторінку суфіксного дерева. Це дозволяє при відлагодженні розпізнавати тип сторінки без потреби інтерпретувати її внутрішню структуру.

- **flags** - містить прапорці, що визначають роль сторінки: коренева, внутрішня, сторінка даних (TID). Це дає змогу застосовувати спеціалізовану логіку під час обходу та оброблення індексу.

Попри отримане рішення 11, низка проєктних аспектів залишається складною й потребує додаткового осмислення. Однією з найбільш проблемних зон є взаємодія з механізмом VACUUM і оброблення видалень. Коли рядок у heap видаляється, його TID необхідно вилучити з усіх вузлів індексу, де він зберігається. Це означає або повне сканування структури індексу, або підтримку зворотних посилань від TID до відповідних вузлів. Обидва підходи мають суттєві недоліки: перший пов'язаний із високими обчислювальними витратами, другий — ускладнює структуру індексу та збільшує накладні витрати на зберігання й супровід додаткових метаданих. Іншим складним напрямом є стиснення. Мітки ребер у суфіксному дереві часто мають спільні префікси, оскільки вони є суфіксами тих самих або подібних рядків. Теоретично можливо застосовувати словникове стиснення або схеми на основі посилань для зменшення обсягу зберігання. Однак такі оптимізації ускладнюють логіку вставки, розщеплення ребер і журналювання змін у WAL, що своєю чергою, збільшує складність реалізації та підвищує ризик помилок. Питання паралелізму також є нетривіальним. Хоча PostgreSQL підтримує паралельну побудову індексів, класичний алгоритм Укконена по суті є послідовним — через залежності, пов'язані з підтримкою суфіксних посилань. Це обмежує можливість прямого використання механізмів паралельного виконання. Альтернативні підходи, зокрема, алгоритми паралельної побудови суфіксних масивів, потенційно могли б забезпечити кращу масштабованість, однак потребують докорінно іншої організації індексної структури. Нарешті, слід враховувати обмеження механізму index-only scan. У PostgreSQL цей режим дає змогу уникати звернень до heap, якщо індекс містить усі необхідні для запиту стовпці. У випадку суфіксного дерева, яке індексує лише текстовий стовпець, повернення самого тексту без звернення до heap вимагало б зберігання його в самому індексі, що ускладнює завдання зменшення обсягу структури.

Лістинг 6: Структура спеціального простору сторінки

```
typedef struct STreeNodePage-
OpaqueData {
    BlockNumber parentNode;
    BlockNumber suffixLink;
    BlockNumber
    BlockNumber prevSiblingNode;
    BlockNumber nextSiblingNode;
    uint16      streePageId;
    uint16      flags;
} STreeNodePageOpaqueData;
```

Висновки

Реалізація суфіксного дерева як індексної структури в PostgreSQL передбачає інтеграцію теоретично елегантної моделі в складну, багаторівневу архітектуру промислової СУБД. У цьому процесі кожне проєктне рішення, пов'язане з необхідністю балансувати між просторовою ефективністю, часовими характеристиками операцій, складністю реалізації

та вимогами до транзакційної надійності й відновлення після збоїв.

Проаналізовані питання — ідентифікація сторінок, організація внутрішньої архітектури сторінки, відображення логічних вузлів на фізичні сторінки, зберігання міток, розміщення TID, взаємодія з буферним менеджером репрезентують лише частину спектру рішень, які необхідно ухвалити. Повноцінна реалізація неминуче потребує десятків додаткових виборів на більш дрібному рівні, кожен з яких впливає на загальну поведінку системи.

Водночас кожна система рівня складності сучасної СУБД фактично формує власний внутрішній програмний каркас — набір інтерфейсів, інваріантів і протоколів взаємодії, які визначають, як саме окремі компоненти можуть співпрацювати. Цей каркас не є нейтральним: він задає допустимі способи розширення, обмежує архітектурні рішення та формує характер компромісів, які доводиться приймати розробникові. Тому впровадження нової індексної структури — це не лише адаптація алгоритму до конкретного програмного інтерфейсу, а й узгодження його з глибинною логікою взаємодії підсистем.

Література

1. Comer D. Ubiquitous B-Tree. *ACM Comput. Surv.* 11. 1979. Vol. 11., no.2. P.121–37. URL: https://doi.org/10.1145/356770.356776. (date of access: 08.05.2025)
2. Hash Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/hash-index.html. (date of access: 08.05.2025)
3. GiST Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/gist.html. (date of access: 08.05.2025)
4. Walid A.G., Ilyas I. F. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems*. 2001. Vol. 11., no.2. P.215–40. URL: https://doi.org/10.1023/A:1012809914301. (date of access: 25.01.2026)
5. SP-GiST Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/spgist.html. (date of access: 08.05.2025)
6. GIN Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/gin.html. (date of access: 08.05.2025)
7. BRIN Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/brin.html. (date of access: 08.05.2025)
8. Зважій Д. Особливості Індексації у PostgreSQL. *Наукові Записки НаУКМА. Комп'ютерні Науки.* 2025. *№*. 8. С. 113–17. DOI: 10.18523/2617-3808.2025.8.113-117. (дата звернення: 25.01.2026)
9. Basic API Structure for Indexes. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/17/index-api.html. (date of access: 08.05.2025)
10. Stonebraker M., Rowe L. A. The design of POSTGRES. *SIGMOD.* 1986. Vol. 15., no. 2 P. 340—355. DOI: 10.1145/16856.16888. URL: https://dl.acm.org/doi/10.1145/16856.16888. (дата зверн. 25.01.2026)
11. Suffix Tree Based AM Index Method Source Code. URL: https://github.com/Uaman/postgres/tree/feature-string-search-am. (date of access: 10.02.2026)
12. Hellerstein J. M., Naughton J. F., Pfeffer A. Generalized Search Trees for Database Systems. *Encyclopedia of Database Systems* Morgan Kaufmann Publishers Inc. 1998. P.1-3. URL: https://doi.org/10.1007/978-1-4899-7993-3_743-2 (date of access: 15.02.2026)
13. Eltabakh M., Ramy E., Walid A. Space-Partitioning Trees in PostgreSQL: Realization and Performance. 2006. URL: https://doi.org/10.1109/ICDE.2006.146. (date of access: 08.05.2025)
14. Schoemans M., Walid G. A., Zimányi E., Sakr M. Multi-Entry Generalized Search Trees for Indexing Trajectories. *Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems. 2024.* P.21–31. URL: https://doi.org/10.1145/3678717.3691320. (date of access: 15.02.2026)
15. Weiner P. Linear Pattern Matching Algorithm. 1973. URL: https://doi.org/10.1109/SWAT.1973.13. (date of access: 10.02.2026)

16. McCreight E. M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM* . 1976. Vol.23., no.2. P.262–72. URL: https://doi.org/10.1145/321941.321946. (date of access: 15.02.2026)
17. Ukkonen E.. On-Line Construction of Suffix Trees. *Algorithmica.* 1995. Vol.14., no.3 P.249–60. URL: https://doi.org/10.1007/BF01206331. (date of access: 10.02.2026)
18. Manber U., Gene M. Suffix Arrays: A New Method for on-Line String Searches. *SIAM Journal on Computing*. 1993. Vol. 22., no. 5: P.935–48. URL: https://doi.org/10.1137/0222058. (date of access: 15.02.2026)
19. Xu W., Haoyu C., Yidong H., Xuedong H., Ge N. Full-Text Search Engine with Suffix Index for Massive Heterogeneous Data. *Information Systems.* 2022. URL: https://doi.org/10.1016/j.is.2021.101893. (date of access: 10.02.2026)
20. Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. *Cambridge University Press*. 1997. URL: https://doi.org/10.1017/CBO9780511574931. (date of access: 08.05.2025)

Дата першого надходження до видання: 17.02.2026
Внутрішня рецензія отримана: 25.02.2026
Зовнішня рецензія отримана: 25.02.2026
Дата прийняття статті до друку: 19.03.2026
Дата публікації: 16.04.2026

Про авторів:

Зважій Дмитро Володимирович,
аспірант,
старший викладач кафедри мережних технологій факультету інформатики,
Zvazhii Dmytro,
Post-graduate student, senior tutor
https://orcid.org/0000-0003-1705-3590
E-mail: d.zvazhii@ukma.edu.ua

Гороховський Семен Самуїлович,
кандидат фізико-математичних наук,
доцент кафедри інформатики факультету інформатики,
Gorokhovsky Semen,
Ph.D (physical and mathematical sciences),
associate professor
E-mail: gor@ukma.edu.ua

Місце роботи авторів:

Національний університет
«Києво-Могилянська академія»,
04655, Україна, Київ,
вул. Григорія Сковороди 2,
National University
“Kyiv–Mohyla Academy”,
faculty of informatics
E-mail: fin@ukma.edu.ua