КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023
DOI 10.28925/2663-4023.2026.33.1124
УДК 004.22

Вічепольський Юрій Юрійович
студент факультету інформаційних технологій і математики
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0009-0007-9227-367X
Vichepolskyi.Yurii2025@vnu.edu.ua

Булатецька Леся Віталіївна
кандидат фіз.-мат. наук, доцент, доцент кафедри комп'ютерних наук та кібербезпеки
Волинський національний університет імені Лесі Українки, Луцьк, Україна
ORCID: 0000-0002-7202-826X
Bulatetska.Lesya@vnu.edu.ua

ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ РЕЛЯЦІЙНИХ БАЗ ДАНИХ ШЛЯХОМ КОМБІНОВАНОГО ВИКОРИСТАННЯ МЕХАНІЗМІВ СЕКЦІОНУВАННЯ ТА ІНДЕКСУВАННЯ

Анотація. У роботі проведено комплексне дослідження методів підвищення швидкодії реляційних систем керування базами даних шляхом інтеграції механізмів секціонування та індексування. Актуальність роботи зумовлена стрімким зростанням обсягів великих даних, що потребує пошуку оптимального балансу між складністю архітектурних рішень і апаратними витратами. Методологія дослідження ґрунтується на серії експериментів у середовищі СУБД PostgreSQL 18.1. Для забезпечення об'єктивності результатів згенеровано синтетичний набір даних обсягом 1 мільйон записів, який за структурою та бізнес-логікою імітує сучасну E-commerce систему. Експериментальна архітектура включає несекціоновану таблицю (Plain) та три типи секціонованих структур: за діапазоном дат (Range), списком категоріальних ознак (List) і хешем ідентифікатора (Hash). Програма випробувань охоплює шість ключових сценаріїв, зокрема агрегаційні запити, складні діапазонні вибірки, точкові звернення та операції з'єднання (JOIN). Основними метриками оцінювання визначено час виконання запитів (latency) та інтенсивність використання ресурсів підсистеми введення-виведення (I/O cost), виміряну в одиницях системних буферів. Результати порівняльного аналізу засвідчили, що секціонування без додаткового індексування забезпечує суттєве прискорення (до 11,9 разів для стратегії Range) лише за умови відповідності критерію фільтрації ключу секціонування, що пояснюється використанням механізму partition pruning. Водночас встановлено, що для точкових запитів несекціоновані таблиці з індексами перевершують секціоновані аналоги на 40-60% через відсутність накладних витрат планувальника на аналіз ієрархії секцій. Найвищу ефективність продемонструвало поєднання секціонування з композитними індексами, яке забезпечило зниження ресурсоспоживання до 99,1% у складних багатофакторних запитах. Окремо проаналізовано вплив фрагментації: встановлено, що під час масових агрегацій несекціонована таблиця може працювати приблизно на 20% швидше завдяки зменшенню кількості операцій з файловими дескрипторами. За результатами дослідження сформульовано практичні рекомендації щодо вибору стратегій оптимізації залежно від профілю навантаження. Робота підтверджує, що секціонування та індексація не є взаємозамінними, а виступають взаємодоповнюючими технологіями, максимальна ефективність яких досягається лише за умови їх узгодженого застосування в межах єдиної архітектури.

Ключові слова: реляційні СУБД; PostgreSQL; секціонування даних; індексування; Range Partitioning; List Partitioning; Hash Partitioning.

ВСТУП

Аналіз останніх досліджень і публікацій. Ефективне зберігання та оперативний доступ до інформації є однією з фундаментальних задач проєктування сучасних інформаційних систем. Саме правильно обрана архітектура бази даних і продумане проєктування її схеми значною мірою визначають успіх програмного продукту. Серед важливих інструментів розробника особливе місце посідає секціонування [1-4]. Найчастіше використовується горизонтальне секціонування, що реалізується за

допомогою підходів Range, List і Hash. Його доцільно застосовувати у випадках, коли обсяг таблиці перевищує 2 ГБ або коли вона зберігає історичні дані. Попри технічні переваги секціонування, його ефективність критично залежить від налаштування доступу до даних. Це змушує інженерів шукати компроміс між складністю архітектури та реальною продуктивністю, що підтверджує необхідність глибокого аналізу перед вибором конкретної стратегії. У роботі [5] показано, що горизонтальне секціонування скорочує час відгуку завдяки мінімізації операцій введення-виведення, але його ефективність прямо залежить від обсягу даних. Для таблиць помірного розміру приріст продуктивності може бути незначним і не виправдовувати накладні витрати на підтримку секцій. Автори роботи [5] підкреслюють відсутність універсального рішення між стратегіями Range та List, оскільки кожна ефективна лише для специфічних типів атрибутів. Прийняття рішення про секціонування має базуватися на аналізі характеру запитів, операцій оновлення та коректному виборі ключів. Секціонування не є абсолютною гарантією прискорення, а вимагає ретельного проєктування з урахуванням реального середовища експлуатації. У роботі [6] проведено детальний аналіз трьох фундаментальних методів секціонування, що спрямовані на мінімізацію затримок при виконанні запитів у високонавантажених системах. Автори приділяють особливу увагу стратегіям складеного (комбінованого) розбиття, зокрема поєднанню секціонування за датами, діапазонами та хеш-функціями. Результати роботи демонструють високу ефективність впровадження гібридних архітектур для оптимізації доступу до даних, що підтверджує доцільність використання складних схем секціонування для підвищення загальної продуктивності баз даних.

У роботі [7] представлено глибокий аналіз функціонування СУБД Oracle в умовах опрацювання 50 мільйонів записів, що дозволяє оцінити реальний вплив секціонування на продуктивність системи. Автори цієї роботи підтверджують, що секціонування радикально оптимізує запити за рахунок обмеження області сканування лише відповідними сегментами таблиці, що дозволяє досягти приросту продуктивності понад 50%. Особливий інтерес становлять висновки дослідників щодо конкуренції між традиційним індексуванням та секціонуванням. Експериментально доведено, що використання індексів є доцільним лише у випадках, коли обсяг результуючої вибірки не перевищує 5% від загальної кількості записів. У сценаріях, де запити оперують більшими масивами даних, стратегія секціонування за атрибутом, що входить до умов фільтрації, виявляється значно ефективнішою. Це узгоджується з висновками авторів роботи [5] про те, що секціонування є критично необхідним саме для надвеликих баз даних.

Автори роботи [8] розглядають проектування ефективних схем секціонування як комплексний модульний процес, наголошуючи на критичній важливості узгодження моделей витрат із характеристиками кластерного середовища та систем зберігання ще до початку етапу розбиття. У дослідженні представлено систематизовану класифікацію стратегій створення розділів за типами алгоритмів, що дозволяє порівнювати їх за критеріями збіжності моделей та якості отриманих секцій.

Окрім секціонування, на продуктивність баз даних істотно впливає використання індексів. У роботі [9] наведено практичні переваги та обмеження різних підходів індексування та рекомендовано обирати стратегію на основі аналізу типових запитів, зокрема застосовувати композитні індекси для багатостовпцевих вибірок і регулярно коригувати їх відповідно до змін навантаження. У досліджені [10] показано, що ефективність індексації залежить від складності запитів, структури та обсягу даних, типу операцій і правильно обраного виду індексу (B-tree, hash, текстові), а також від можливості використання композитних і альтернативних індексів. У роботі [11] розглянуто розширений спектр стратегій індексації (bitmap, full-text, просторові) та підкреслено необхідність врахування витрат на підтримку індексів і особливостей розподілених та хмарних середовищ. Водночас у дослідженні [12] доведено переваги адаптивних методів індексації, зокрема самоналаштовуваних і заснованих на штучному інтелекті індексів, які здатні покращувати швидкодію запитів до 45% та ефективніше працювати в умовах змінних OLTP/OLAP-навантажень. Таким чином, результати [9-12] підтверджують, що обґрунтований і адаптивний вибір стратегій індексування є ключовим чинником підвищення швидкодії та масштабованості систем керування базами даних.

Також важливим напрямом досліджень є взаємодія секціонування з індексними структурами. У контексті динамічної оптимізації баз даних особливої уваги заслуговує розробка системи Shinobi, що дозволяє автоматизувати ці процеси [13]. Автори пропонують підхід горизонтального секціонування в поєднанні з вибірковим індексуванням лише тих сегментів даних, до яких найчастіше звертаються запити [13]. Таке рішення дозволяє суттєво зменшити загальний розмір індексів та витрати на їх підтримку, особливо в умовах нерівномірного навантаження та інтенсивних операцій вставки. Автори отримали 60-кратне покращення продуктивності порівняно з традиційно індексованими таблицями, що підкреслює стратегічну важливість гнучких методів розподілу та індексування даних для сучасних високонавантажених систем.

На основі проведеного аналізу наукових праць стає очевидним, що попри значну кількість теоретичних розробок, питання вибору конкретної комбінації методів оптимізації для специфічних типів запитів залишається відкритим.

Постановка проблеми. Зростання обсягів великих даних призводить до експоненціального збільшення часу виконання SQL-запитів та надмірного споживання ресурсів I/O Buffers. Традиційні методи індексування часто виявляються недостатніми для масивів даних обсягом у мільйони записів, оскільки призводять до створення громіздких структур, які важко підтримувати. Водночас застосування лише механізмів секціонування без належної конфігурації індексів не забезпечує необхідної селективності при виконанні багатофакторних та точкових запитів. Виникає потреба у пошуку оптимальних комбінацій методів розподілу та індексування даних для мінімізації апаратних витрат.

Метою роботи є дослідження ефективності обробки даних у реляційних СУБД при застосуванні комбінованих стратегій горизонтального секціонування (Range, List, Hash) та цільового індексування (одноколонкових, композитних та Bitmap-індексів), а також визначення їхнього впливу на мінімізацію часових затримок і ресурсоспоживання системи.

МЕТОДИКА ДОСЛІДЖЕННЯ

Для проведення експерименту було обрано СУБД PostgreSQL версії 18.1. У якості клієнтського інструменту використовується pgAdmin 4.

Для демонстрації впливу секціонування та індексів на ефективність виконання SQL-запитів у великих реляційних таблицях була створена тестова база даних. Вона містить одну логічну таблицю “замовлень” (orders), яка дублюється в кількох фізичних реалізаціях – кожна має свій тип секціонування або індексну структуру та таблицю «покупців» (customers) для тестування JOIN-запитів (табл. 1).

Структура таблиці orders_plain, яка не використовує механізмів секціонування моделює реальну сутність "Замовлення" в E-commerce системах і містить поля: ідентифікатор, дата замовлення, ідентифікатор клієнта, регіон, сума та статус.

Таблиця 1

Структура тестової бази даних

Таблиця	Тип	Призначення
orders_plain	Звичайна таблиця без секціонування	Базовий варіант для порівняння
orders_range	Секціонована по роках (RANGE)	Розподіл рядків за роками (YEAR(order_date))
orders_list	Секціонована по категоріях (LIST)	Розподіл рядків за регіонами (region)
orders_hash	Секціонована по customer_id (HASH)	Рівномірний розподіл клієнтів по секціях
customers	Довідник покупців	Для тестування JOIN-запитів із замовленнями

```
CREATE TABLE orders_plain (
    id SERIAL PRIMARY KEY,
    order_date DATE NOT NULL,
    customer_id INT NOT NULL,
    region TEXT NOT NULL,
    amount NUMERIC(10, 2),
    status TEXT NOT NULL,
    payload TEXT
     );

CREATE TABLE customers (
     id INT,
     name TEXT
     );
```

Наповнення таблиць здійснено синтетичними даними за допомогою функції generate_series. Обсяг даних таблиці orders_plain становить 1 мільйон записів, а таблиці customers – 1000. Дані таблиці orders_plain були розподілені в часовому діапазоні з 2020 по 2024 рік.

Було реалізовано стратегію секціонування таблиці orders за діапазоном значень. В якості ключа секціонування обрано поле order_date. Створено батьківську таблицю orders_range та 5 дочірніх секцій, кожна з яких налаштована на зберігання даних за один календарний рік (від 2020 до 2024 включно). Всі дані з базової таблиці orders_plain були перенесені у відповідні секції згідно з їхніми часовими мітками.

```
CREATE TABLE orders_range (
       id INT GENERATED BY DEFAULT AS IDENTITY,
       order_date DATE NOT NULL,
       customer_id INT,
       region TEXT,
       amount NUMERIC,
       status TEXT,
       payload TEXT
) PARTITION BY RANGE (order_date);
CREATE TABLE orders_y2020 PARTITION OF orders_range
       FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');
CREATE TABLE orders_y2021 PARTITION OF orders_range
       FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');
CREATE TABLE orders_y2022 PARTITION OF orders_range
       FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');
CREATE TABLE orders_y2023 PARTITION OF orders_range
       FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
CREATE TABLE orders_y2024 PARTITION OF orders_range
     FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

Реалізовано секціонування за списком для розподілу даних за категоріальною ознакою. Ключем секціонування визначено поле region. Структуру даних фізично розділено на три таблиці: orders_list_ua (для значення 'UA'), orders_list_eu ('EU') та orders_list_us ('US'). Дані розподілено між цими таблицями відповідно до значення регіону в кожному рядку.

```
CREATE TABLE orders_list (
       id INT GENERATED BY DEFAULT AS IDENTITY,
       order_date DATE,
       customer_id INT,
       region TEXT NOT NULL,
       amount NUMERIC,
       status TEXT,
       payload TEXT
) PARTITION BY LIST (region);

CREATE TABLE orders_list_ua PARTITION OF orders_list FOR VALUES IN ('UA');
CREATE TABLE orders_list_eu PARTITION OF orders_list FOR VALUES IN ('EU');
CREATE TABLE orders_list_us PARTITION OF orders_list FOR VALUES IN ('US');
```

Застосовано стратегію хеш-секціонування за полем id. Батьківську таблицю налаштовано на поділ за модулем (modulus) 4. Створено 4 дочірні таблиці-секції (orders_hash_0 ... orders_hash_3). При вставці даних СУБД автоматично розподіляє рядки між секціями на основі обчисленого хешу від значення ідентифікатора.

```
CREATE TABLE orders_hash (
       id INT GENERATED BY DEFAULT AS IDENTITY,
       order_date DATE,
       customer_id INT,
       region TEXT,
       amount NUMERIC,
       status TEXT,
       payload TEXT
) PARTITION BY HASH (id);

CREATE TABLE orders_hash_0 PARTITION OF orders_hash FOR VALUES WITH (MODULUS 4,
REMAINDER 0);
CREATE TABLE orders_hash_1 PARTITION OF orders_hash FOR VALUES WITH (MODULUS 4,
REMAINDER 1);
CREATE TABLE orders_hash_2 PARTITION OF orders_hash FOR VALUES WITH (MODULUS 4,
REMAINDER 2);
     CREATE TABLE orders_hash_3 PARTITION OF orders_hash FOR VALUES WITH (MODULUS
4, REMAINDER 3);
```

Для комплексного аналізу ефективності виконання операцій було проведено серію запитів до тестових таблиць. Ключовою особливістю тестування є порівняльний аналіз: запити виконувалися як на базових таблицях (без секціонування Plain), так і на структурах із різними методами розділення даних – за діапазоном (Range), списком (List) та хешем (Hash), а також із застосуванням різних стратегій індексування. Такий підхід дозволяє об'єктивно оцінити вплив архітектурних рішень на швидкість обробки даних при різних сценаріях навантаження. В табл. 2. наведено перелік ключових сценаріїв тестування, що охоплюють операції базової агрегації, точкові вибірки та об'єднання таблиць.

Для отримання об'єктивних даних щодо впливу архітектури таблиць на швидкість обробки даних було проведено серію вимірювань. Основними критеріями оцінки стали час виконання запитів (latency) та витрати ресурсів системи вводу-виводу (I/O Cost). Останній показник вимірювався в одиницях системних буферів (Buffers), що дозволяє оцінити реальну інтенсивність звернення до дискової підсистеми незалежно від поточного завантаження процесора.

Таблиця 2

Перелік ключових сценаріїв тестування

№	Тип запиту	Формулювання запиту	Запит
1	Простий SELECT (COUNT)	Запит для підрахунку загальної кількості рядків у вихідній таблиці.	`SELECT count(*) FROM orders;`
2	Фільтр по даті (Range Query)	Пошук всіх замовлень за конкретну дату.	`SELECT * FROM orders WHERE order_date='2022-05-15';`
3	Фільтр по регіону (Category Query)	Обчислення середнього чека для регіону 'UA'.	`SELECT avg(amount) FROM orders WHERE region = 'UA';`
4	Фільтр по ID (Point Lookup)	Пошук замовлення за ідентифікатором.	`SELECT * FROM orders WHERE id = 500000;`
5	Агрегатний запит (SUM по всій таблиці)	Розрахунок загального обороту компанії.	`SELECT sum(amount) FROM orders;`
6	Використання оператора JOIN для з'єднання таблиць за зовнішнім ключем	Отримання списку замовлень із іменами клієнтів.	`SELECT o.id, c.name FROM orders_plain o JOIN customers c ON o.customer_id = c.id WHERE o.order_date = '2022-05-15'.`

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ ДОСЛІДЖЕННЯ

Аналіз ефективності методів секціонування без індексування. Тестування проводилося в умовах відсутності індексів, щоб виокремити чистий ефект від використання різних механізмів секціонування: Range (за діапазоном дат), List (за списком регіонів) та Hash (за хешем ідентифікатора). У табл. 3 та табл. 4 наведено порівняльні дані, які демонструють переваги кожного з підходів для конкретних типів бізнес-логіки. На основі отриманих емпіричних даних (табл. 3, табл. 4) сформульовано висновки щодо вибору стратегії секціонування та впливу архітектури на продуктивність.

Вибір методу секціонування повинен базуватися на аналізі профілю навантаження, а саме на тому, які атрибути найчастіше зустрічаються у блоці WHERE.

Метод секціонування за діапазоном (Range Partitioning) є найбільш ефективною стратегією при роботі з часовими рядами та упорядкованими числовими масивами. Доцільність застосування даної архітектури підтверджується результатами проведених випробувань. У сценаріях із фільтрацією за часовим показником та при виконанні операцій з'єднання (JOIN) було зафіксовано прискорення обробки запитів до 11.9 разів паралельно зі зниженням навантаження на підсистему вводу-виводу (I/O) на 80%. Висока продуктивність механізму зумовлена лінійною залежністю між селективністю запиту та обсягом сканованих даних. Зокрема, при вибірці даних за один рік із загального п'ятирічного масиву система опрацьовує рівно 20% від загального об'єму інформації, що дозволяє суттєво оптимізувати обчислювальні ресурси.

Метод секціонування за списком (List Partitioning) є оптимальним для роботи з даними що мають чітку категоріальну приналежність та обмежену кількість унікальних значень. Ця архітектура демонструє найкращі результати при сортуванні за географічними регіонами або статусами замовлень та ідентифікаторами філій.

Таблиця 3

Порівняльний аналіз часу виконання запитів без індексів (ms)

№	Тип запиту	Plain	Range	List	Hash	Кращий
1	COUNT *	119.4	119.1	120.6	118.3	-
2	WHERE (date)	120.6	**10.1**	115.5	124.9	Range (11.9x)
3	WHERE (region)	60.4	66.5	**46.2**	82.4	List (1.3x)
4	WHERE (id)	42.4	43.8	39.1	**34.6**	Hash (1.2x)
5	Агрегація (sum)	**63.9**	80.8	79.1	80.4	Plain
6	JOIN (date)	41.7	**6.6**	40.6	39.8	Range (6.3x)

Таблиця 4

Порівняльний аналіз ресурсоспоживання запитів без індексів (I/O Cost, в одиницях Buffers)

№	Тип запиту	Plain	Range	List	Hash	Економія ресурсів
1	COUNT *	11,364	11,366	11,366	11,366	-
2	WHERE (date)	11,364	**2,272**	11,366	11,366	Range (-80.0%)
3	WHERE (region)	11,364	11,366	**3,747**	11,366	List (-67.0%)
4	WHERE (id)	11,364	11,366	11,366	**2,844**	Hash (-75.0%)
5	Агрегація (sum)	**11,364**	11,366	11,366	11,366	-
6	JOIN (date)	11,382	**2,278**	11,384	11,378	Range (-80.0%)

Проведені тести зафіксували приріст швидкості виконання запитів у 1.3 рази при здійсненні вибірки за конкретним регіоном. Загальна ефективність цього методу безпосередньо залежить від характеру розподілу інформації в системі. Найвищий приріст продуктивності спостерігається у випадках коли на обрану категорію припадає мінімальна частка від загального обсягу даних. Такий підхід дозволяє базі даних ігнорувати невідповідні секції та значно прискорювати процес пошуку необхідних записів.

Метод хеш-секціонування (Hash Partitioning)призначений для забезпечення рівномірного розподілу інформації між фізичними сегментами та оптимізації точкових запитів. Ця стратегія стає незамінною у випадках коли логічні критерії (дата/список) є незастосовними або спричиняють нерівномірне навантаження на систему. Під час практичних випробувань у сценарії пошуку за ідентифікатором цей метод продемонстрував прискорення виконання операцій у 1.2 рази. Одночасно з цим було зафіксовано суттєву економію ресурсів підсистеми вводу-виводу на рівні 75%. Спеціальна хеш-функція дозволяє системі миттєво обчислити розташування цільової секції без необхідності послідовної перевірки діапазонів або списків. Такий механізм демонструє особливу ефективність при роботі з великою кількістю секцій та дозволяє підтримувати стабільно високу швидкість доступу до даних.

Вплив кількості секцій на продуктивність. Архітектура секціонування вносить у роботу СУБД як можливості оптимізації, так і накладні витрати. Зі збільшенням кількості секцій (при незмінному обсязі даних) розмір кожної окремої секції зменшується. Це позитивно впливає на час виконання запитів з високою селективністю. Час виконання $T \approx \frac{T_{total}}{N}$, де N – кількість секцій (за умови ідеального ефекту відсікання). У нашому експерименті поділ на 5 секцій (range) прискорив відповідні запити пропорційно у 5 разів (по I/O). Крім того збільшення кількості секцій створює навантаження на планувальник запитів та файлову систему. У запитах типу Агрегація (SUM) та COUNT *, де необхідно просканувати всі секції, звичайна таблиця виявилася на ~20% швидшою за секціоновані. Причина – операція Parallel Append та необхідність відкриття/закриття дескрипторів файлів для кожної секції додають затримку.

Надмірна фрагментація (тисячі секцій) може деградувати продуктивність глобальних запитів. Рекомендована стратегія – обирати таку кількість секцій, щоб активна секція повністю поміщалася в оперативну пам’ять сервера.

Аналіз комбінованого впливу секціонування та індексів на швидкість обробки даних. Після завершення базового аналізу продуктивності на структурах без додаткової оптимізації було ініційовано етап впровадження індексних структур для всіх типів таблиць. Цей крок спрямований на детальне вивчення взаємодії між механізмами секціонування та різними стратегіями індексування для досягнення максимальної швидкості обробки даних. Процес оптимізації охоплює створення стандартних одноколонкових індексів за первинними ідентифікаторами та часовими показниками для прискорення точкових і діапазонних вибірок.

Паралельно з цим впроваджуються складені композитні індекси за полями регіону та статусу для ефективної обробки багатофакторних запитів. Також застосовується стратегія індексування категоріальних ознак за допомогою bitmap-підходу для оптимізації роботи з полями що мають обмежену

кількість унікальних значень. Комплексне використання цих інструментів як на несекціонованих так і на секціонованих таблицях дозволяє об'єктивно оцінити рівень зниження витрат ресурсів та приросту швидкодії в умовах реальних аналітичних навантажень.

Для практичної реалізації зазначеної стратегії та подальшого порівняльного аналізу було виконано наступний перелік технічних операцій з налаштування індексних структур:

1. створення одноколонкових індексів:
```
CREATE INDEX idx_plain_id ON orders(id);
CREATE INDEX idx_plain_date ON orders(order_date);
```
2. створення складеного (композитного) індексу:
```
CREATE INDEX idx_plain_reg_stat ON orders(region, status);
```
3. створення індексів bitmap:
```
CREATE INDEX idx_plain_status ON orders(status);
```

Після успішного впровадження індексних структур було проведено повторну серію випробувань для оцінки їхнього впливу на швидкість обробки даних та загальне навантаження на систему. Отримані результати дозволяють проаналізувати зміну продуктивності як для стандартних одноколонкових індексів так і для складніших композитних та bitmap-рішень у контексті різних методів секціонування. Основними критеріями оцінки знову виступили час виконання операцій у мілісекундах та обсяг операцій введення-виведення у одиницях системних буферів що забезпечує пряме порівняння з показниками до оптимізації (Табл. 5-10).

Таблиця 5

Порівняльний аналіз часу виконання запитів з вчикористанням звичайних індексів (ms)

№	Тип запиту	Plain	Range	List	Hash	Кращий
1	COUNT *	108.2	114.5	112.8	111.0	-
2	WHERE (date)	0.42	**0.34**	0.62	0.50	Range (1.2x)
3	WHERE (region)	56.5	62.4	**42.0**	60.8	List (1.35x)
4	WHERE (id)	**0.032**	0.053	0.042	0.036	Plain (1.6x)
5	Агрегація (sum)	**63.0**	76.3	83.2	77.0	Plain
6	JOIN (date)	**0.57**	**0.59**	0.67	0.81	Plain / Range

Таблиця 6

Порівняльний аналіз ресурсоспоживання запитів з використанням звичайних індексів (I/O Cost, в одиницях Buffers)

№	Тип запиту	Plain	Range	List	Hash	Економія ресурсів
1	COUNT *	900	909	874	**858**	Hash (-5%)
2	WHERE (date)	552	**499**	559	559	Range (-10%)
3	WHERE (region)	11,364	11,366	**3,747**	11,366	List (-67%)
4	WHERE (id)	**4**	16	10	**4**	Plain / Hash
5	Агрегація (sum)	11,364	11,366	11,366	11,366	-
6	JOIN (date)	564	**511**	565	571	Range (-9%)

Таблиця 7

Порівняльний аналіз часу виконання запитів з використанням композитних індексів (ms)

Тип запиту	Plain	Range	List	Hash	Кращий
WHERE reg='UA' AND stat='PAID'	**6.25**	10.93	6.92	11.17	Plain

Таблиця 8

Порівняльний аналіз ресурсоспоживання запитів з використанням композитних індексів (I/O Cost, в одиницях Buffers)

Тип запиту	Plain	Range	List	Hash	Економія ресурсів
WHERE reg='UA' AND stat='PAID'	97	108	**96**	108	~99% (було 11,366)

Таблиця 9

Порівняльний аналіз часу виконання запитів з використанням bitmap (ms)

Тип запиту	Plain	Range	List	Hash	Кращий
WHERE status='NEW'	**9.1**	36.0	36.5	37.2	Plain (4x)

Таблиця 10

Порівняльний аналіз ресурсоспоживання запитів з використанням bitmap (I/O Cost, в одиницях Buffers)

Тип запиту	Plain	Range	List	Hash	Економія ресурсів
WHERE status='NEW'	**144**	154	150	154	Plain

Тестування простих B-Tree індексів показало, що для точкових запитів (WHERE id = ...) несекціонована таблиця працює швидше за секціоновані (0.032 ms проти 0.053 ms у Range) (табл. 3, табл 4.). Причина полягає в тому, секціонування вносить незначні накладні витрати на планування запиту та вибір відповідної секції. У звичайній таблиці шлях до даних прямий: Індекс -> Рядок.

Використання композитного індексу (region, status) дозволило досягти найвищої продуктивності (6-11 ms для підрахунку 100 тис. рядків) (табл. 5, табл. 6).

СУБД не зверталася до таблиці даних (Heap Fetches: 0), отримуючи всю інформацію виключно зі структури індексу. Це підтверджує, що "покриваючі індекси" є найефективнішим методом оптимізації як для звичайних, так і для секціонованих таблиць.

У тесті з низькою селективністю (WHERE status = 'NEW') (таб. 7, табл. 8), де критерій пошуку не співпадав з ключем секціонування, звичайна таблиця виявилася у 4 рази швидшою за секціоновані (9 ms проти ~36 ms). Так як планувальник був змушений сканувати індекси кожної окремої секції та об'єднувати результати, тоді як у Plain таблиці це була одна атомарна операція.

Завершальний етап дослідження передбачає проведення інтегрального порівняння всіх випробуваних конфігурацій бази даних відносно базової несекціонованої таблиці без індексних структур. Цей аналіз дозволяє чітко розмежувати внесок механізмів секціонування та засобів індексування у загальне підвищення швидкодії системи при виконанні різних сценаріїв запитів. У ході тестування було зіставлено результати для класичних архітектур та секціонованих масивів як у чистому вигляді так і з застосуванням оптимізаційних індексів.

Особлива увага приділяється виявленню граничних показників продуктивності для складних фільтрів та масових вибірок де поєднання композитних індексів із логічним розподілом даних демонструє найбільш радикальні зміни у швидкості обробки запитів. Наведені у табл. 11 статистичні дані відображають зміну часу виконання від початкових показників до найбільш оптимізованих значень та дозволяють ідентифікувати найкращий технологічний стек для кожного конкретного типу бізнес-логіки. Табл. 12 демонструє фінальні результати тестування та слугує об'єктивним підтвердженням ефективності обраної стратегії модернізації бази даних.

Таблиця 11

Зведена інформація щодо часу виконання запитів (Execution Time, ms) порівняно з базовою несекціонованою таблицею без індексів

Сценарій запиту	Plain (Без індексів)	Partitioning (Без індексів)	Plain + Index	Partitioning + Index	Найкращий результат
Пошук за Датою (Range)	120.60	10.10	0.42	**0.34**	Range + Index
Пошук за Регіоном (List)	60.40	46.20	-	**6.92***	List + Index
Точковий пошук ID (Hash)	42.40	34.60	**0.03**	0.04	Plain + Index
Складний фільтр (UA+PAID)	60.00	46.00	**6.25**	6.92	Plain + Index
Масова вибірка (Status=NEW)	65.00	80.00	**9.10**	36.00	Plain + Index

***** – для пошуку за регіоном у стовпчику "Index" використано композитний індекс.

Таблиця 12

Економія системних ресурсів (I/O Buffers) Кількість зчитаних сторінок пам'яті (8KB blocks)

Сценарій запиту	Plain (Без індексів)	Partitioning + Index (Best Case)	Економія ресурсів
Пошук за Датою	11,364	499	95.6%
Точковий пошук ID	11,364	4	99.9%
Складний фільтр	11,364	96	99.1%
Масова вибірка	11,364	144	98.7%

Експериментально підтверджено, що секціонування саме по собі не є повноцінною заміною індексації. Максимальні показники продуктивності системи досягаються виключно завдяки поєднання двох механізмів, які діють на різних рівнях абстракції. Секціонування працює на логічному рівні, виконуючи роль грубого фільтра, що відсікає значні масиви даних (гігабайти інформації), які гарантовано не відповідають умовам запиту. Індексація ж вступає в дію на фізичному рівні вже всередині відфільтрованої секції, дозволяючи миттєво локалізувати конкретний рядок і мінімізуючи кількість операцій вводу-виводу (I/O).

Архітектурною перевагою PostgreSQL є використання локальних секціонованих індексів, які будуються окремо для кожної фізичної секції. Такий підхід дозволяє зменшити розмір кожного окремого B-Tree дерева, що підвищує ефективність його обробки в оперативній пам'яті та покращує кешування. Крім того, це забезпечує гнучкість в управлінні життєвим циклом даних. Видалення застарілих секцій (DROP TABLE partition) відбувається миттєво разом із відповідною частиною індексу, не створюючи блокувань для всієї системи, що є критичним для високонавантажених баз даних.

Критичною умовою ефективності є узгодженість ключів індексації зі стратегією секціонування. Індекс повинен корелювати з критерієм поділу таблиці. Наприклад, для таблиці, розділеної за датою, створення індексу за тим же полем є оптимальним рішенням. Натомість індексація за незалежним полем, таким як customer_id, призведе до того, що індекс буде "розмазаний" по всіх фізичних файлах. Це змусить СУБД сканувати секції за всі роки для пошуку одного клієнта, нівелюючи переваги секціонування.

Окрему увагу слід приділяти компенсації накладних витрат (overhead). Оскільки робота зі складною ієрархією таблиць вимагає від планувальника запитів додаткового часу на аналіз метаданих, рекомендується застосовувати більш агресивну стратегію індексації. Зокрема, використання композитних індексів дозволяє прискорити виконання запиту всередині секції настільки, щоб повністю компенсувати час, витрачений на вибір цієї секції, забезпечуючи загальний виграш у продуктивності.

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

На основі проведеного теоретичного аналізу та практичних експериментів, сформульовано наступні рекомендації для розробників та адміністраторів баз даних (табл. 13).

Вибір типу секціонування. У табл. 13 систематизовано ключові сценарії використання архітектурних рішень для забезпечення максимальної ефективності роботи систем керування базами даних у різних експлуатаційних умовах.

Таблиця 13

Рекомендації щодо використання секціонування

Тип секціонування	Рекомендований сценарій	Приклад використання
RANGE	Робота з часовими рядами, історичними даними, де запити часто оперують періодами (місяць, рік). Ідеально для реалізації політики видалення старих даних.	Логи подій, фінансові транзакції, історія замовлень.
LIST	Дані мають чітку категоріальну приналежність з обмеженою кількістю значень. Ефективно для мультитенантних систем (SaaS).	Дані по філіях, регіонах країни, статусах (якщо їх багато).
HASH	Рівномірний розподіл великих масивів даних без логічного критерію групування. Використовується для запобігання "гарячих точок" при запису.	Таблиці користувачів, сесій, товарів у глобальних каталогах.

Вибір індексів. Результати тестування простих B-Tree індексів показали цікаву залежність між архітектурою таблиці та швидкістю точкових запитів. Експериментально доведено, що для пошуку поодиноких записів за ідентифікатором звичайна несекціонована таблиця працює дещо швидше за секціоновані аналоги (0.032 мс проти 0.053 мс). Це пояснюється наявністю додаткових накладних витрат (overhead) у секціонованих таблицях, де планувальник запитів змушений витрачати ресурс процесора на вибір відповідної секції перед зверненням до індексу. У монолітній таблиці шлях до даних є прямим, що забезпечує мінімальну латентність.

Окремої уваги заслуговує ефективність композитних індексів, які в ході експерименту дозволили досягти режиму Index Only Scan. При використанні індексу, що охоплює всі поля запиту (у нашому випадку регіон та статус), СУБД отримала можливість повертати результат без звернення до основної таблиці даних. Це призвело до нульової кількості зчитувань сторінок даних (Heap Fetches: 0) та

максимальної продуктивності системи. Такий результат підтверджує, що використання покриваючих індексів є найефективнішим методом оптимізації, незалежно від того, чи застосовується секціонування.

Водночас, тестування виявило вразливість секціонування при глобальному пошуку за неключовими полями. У сценаріях з низькою селективністю (наприклад, пошук за статусом NEW), де критерій фільтрації не збігається з ключем секціонування, продуктивність секціонованих таблиць виявилася в 4 рази нижчою за звичайну таблицю. Це зумовлено необхідністю сканувати індекси кожної окремої секції та виконувати ресурсоємну операцію об'єднання результатів, тоді як у звичайній таблиці ця операція є атомарною.

Комбіноване використання секцій та індексів. Проведене дослідження дозволяє стверджувати, що секціонування та індексація не є взаємозамінними технологіями, а найкращі результати досягаються виключно при їх поєднанні. Секціонування працює на логічному рівні, відсікаючи значні масиви даних (гігабайти інформації), які гарантовано не відповідають умовам запиту, тоді як індекси працюють на фізичному рівні, забезпечуючи миттєву навігацію всередині вже відфільтрованого обсягу. Саме поєднання механізмів Partition Pruning (відсікання секцій) та Index Scan забезпечило рекордне прискорення запитів у 12 разів під час тестів.

Ключовою перевагою використання індексів у середовищі секціонування PostgreSQL є їхня локальність. Оскільки індекси будуються окремо для кожної фізичної секції, вони мають значно менший розмір порівняно з глобальним індексом монолітної таблиці. Компактність локальних індексів підвищує ймовірність їх повного кешування в оперативній пам'яті, що критично зменшує кількість дорогих операцій вводу-виводу (I/O). Крім того, це спрощує адміністрування, дозволяючи видаляти застарілі дані разом з їхніми індексами миттєво, не блокуючи роботу всієї бази даних.

Однак для досягнення максимальної ефективності критично важливою є стратегія компенсації накладних витрат. Оскільки сама структура секціонування додає певну затримку при плануванні запитів, рекомендується використовувати більш агресивні стратегії індексації, зокрема композитні індекси. Експеримент підтвердив, що індекс, який починається з поля з високою кардинальністю або поля рівності, здатен нівелювати затримки планувальника, роблячи роботу із секціонованою таблицею такою ж швидкою, як і з монолітною, але при цьому зберігаючи всі переваги масштабованості.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dhandala, N. (2026, January 30). *How to build database partitioning types*. OneUptime. https://oneuptime.com/blog/post/2026-01-30-database-partitioning-types/view
2. Jex, K. (2025, September 19). *Postgres partitioning best practices: Sofia's story*. https://karenjex.blogspot.com/2025/09/postgres-partitioning-best-practices.html
3. Angell, Z. (2023, November 2). *Database partitioning best practices*. Prefect. https://www.prefect.io/blog/database-partitioning-prod-postgres-without-downtime
4. Омельчук, А., Юзва, А., & Булатецька, Л. (2024, June 13-16). Методи високоефективної обробки даних в Oracle за допомогою секціонування таблиць та індексів. In *Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем: матеріали І міжнар. науково-практ. конф.* (p. 48). Луцьк. https://apcssm.vnu.edu.ua/index.php/conf/article/view/54
5. Matalqa, S., & Mustafa, S. (2015, December 15-17). *The effect of horizontal database table partitioning on query performance* [Paper presentation]. International Arab Conference on Information Technology (ACIT'2015). https://acit2k.org/ACIT/index.php/proceedings-acit/acit-2015-proceedings
6. Sawant, M., Kinage, K., Pilankar, P., & Chaudhari, N. (2013). Database partitioning: A review paper. *International Journal of Innovative Technology and Exploring Engineering*, 3(5), 82-85. https://www.ijitee.org/portfolio-item/E1259103513/
7. Ajdari, J., Mustafa, N., Zenuni, X., et al. (2016). Impact of table partitioning on the query execution performance. *IJCSI International Journal of Computer Science Issues*, 13(4), 52-58. https://www.ijcsi.org/papers/IJCSI-13-4-52-58.pdf
8. Liu, P.-J., Li, C.-P., & Chen, H. (2024). Enhancing storage efficiency and performance: A survey of data partitioning techniques. *Journal of Computer Science and Technology*, 39, 346-368. https://doi.org/10.1007/s11390-024-3538-1
9. Anchlia, A. (2024). Enhancing query performance through relational database indexing. *International Journal of Computer Trends and Technology (IJCTT)*, 72(8), 130-133. https://doi.org/10.14445/22312803/IJCTT-V72I8P119
10. Голубінка, В., & Худий, А. (2024). Підвищення продуктивності запитів до баз даних: аналіз технік індексації. *Вісник Національного університету «Львівська політехніка». Серія: Інформаційні системи та мережі*, (15), 65-73.

11. Azmat, H., & Huma, Z. (2025). Indexing strategies in SQL: Enhancing query efficiency and scalability. *Baltic Journal of Multidisciplinary Research*, 2(2), 130-138. https://balticpapers.com/index.php/bjmr/article/view/53
12. Robinson, E., & Anderson, J. (2025). Comparative study of adaptive indexing techniques for performance improvement in dynamic workloads. *The Journal of Innovation in Governance and Business Practices*, 1, 32–58. https://jigbp.com/index.php/jigbp/article/view/2
13. Wu, E., & Madden, S. (2011). Partitioning techniques for fine-grained indexing. In *Proceedings of the International Conference on Data Engineering (ICDE)* (pp. 1126-1137). http://hdl.handle.net/1721.1/63110

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023

Yurii Vichepolskyi
Student of the Faculty of Informations Technologies and Mathematics
Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID: 0009-0007-9227-367X
Vichepolskyi.Yurii2025@vnu.edu.ua

Lesia Bulatetska
PhD, Associate Professor, Associate Professor at the Department of Computer Science and Cybersecurity, Lesya Ukrainka Volyn National University, Lutsk, Ukraine
ORCID: 0000-0002-7202-826X
Bulatetska.Lesya@vnu.edu.ua

OPTIMIZATION OF RELATIONAL DATABASE PERFORMANCE THROUGH THE COMBINED USE OF PARTITIONING AND INDEXING MECHANISMS

Abstract. This paper presents a comprehensive study of methods for improving the performance of relational database management systems through the integration of partitioning and indexing mechanisms. The relevance of the study is driven by the rapid growth of big data volumes, which necessitates finding an optimal balance between architectural complexity and hardware costs. The research methodology is based on a series of experiments conducted in the PostgreSQL 18.1 database management system environment. To ensure objective results, a synthetic dataset consisting of 1 million records was generated, simulating the structure and business logic of a modern e-commerce system. The experimental architecture includes a non-partitioned table (Plain) and three types of partitioned structures: date range partitioning (Range), categorical list partitioning (List), and identifier hash partitioning (Hash). The testing framework covers six key query scenarios, including aggregation queries, complex range-based selections, point lookups, and join operations. The primary evaluation metrics are query execution time (latency) and input/output subsystem resource utilization (I/O cost), measured in units of system buffers. The comparative analysis demonstrates that partitioning without additional indexing provides significant performance improvements (up to 11.9× for the Range strategy) only when the query filtering condition matches the partitioning key, which is achieved through the partition pruning mechanism. At the same time, it was found that for point queries, non-partitioned tables with indexes outperform partitioned counterparts by 40-60%, due to the absence of planner overhead associated with analyzing the partition hierarchy. The highest efficiency was achieved by combining partitioning with composite indexes, resulting in up to a 99.1% reduction in resource consumption for complex multi-factor queries. The impact of data fragmentation was also analyzed, revealing that during large-scale aggregation queries, a non-partitioned table may perform approximately 20% faster due to a reduced number of file descriptor operations. Based on the findings, practical recommendations are formulated for selecting optimization strategies depending on workload characteristics. The study confirms that partitioning and indexing are not interchangeable but complementary technologies, whose maximum potential is realized only through their coordinated application within a unified database architecture.

Keywords: relational DBMS; PostgreSQL; data partitioning; indexing; Range Partitioning; List Partitioning; Hash Partitioning.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Dhandala, N. (2026, January 30). *How to build database partitioning types*. OneUptime. https://oneuptime.com/blog/post/2026-01-30-database-partitioning-types/view
2. Jex, K. (2025, September 19). *Postgres partitioning best practices: Sofia's story*. https://karenjex.blogspot.com/2025/09/postgres-partitioning-best-practices.html
3. Angell, Z. (2023, November 2). *Database partitioning best practices*. Prefect. https://www.prefect.io/blog/database-partitioning-prod-postgres-without-downtime
4. Омельчук, А., Юзва, А., & Булатецька, Л. (2024, June 13-16). Методи високоефективної обробки даних в Oracle за допомогою секціонування таблиць та індексів. In *Проблеми комп'ютерних наук, програмного моделювання та безпеки цифрових систем: матеріали І міжнар. науково-практ. конф.* (p. 48). Луцьк. https://apcssm.vnu.edu.ua/index.php/conf/article/view/54

5. Matalqa, S., & Mustafa, S. (2015, December 15-17). *The effect of horizontal database table partitioning on query performance* [Paper presentation]. International Arab Conference on Information Technology (ACIT'2015). https://acit2k.org/ACIT/index.php/proceedings-acit/acit-2015-proceedings
6. Sawant, M., Kinage, K., Pilankar, P., & Chaudhari, N. (2013). Database partitioning: A review paper. *International Journal of Innovative Technology and Exploring Engineering*, 3(5), 82-85. https://www.ijitee.org/portfolio-item/E1259103513/
7. Ajdari, J., Mustafa, N., Zenuni, X., et al. (2016). Impact of table partitioning on the query execution performance. *IJCSI International Journal of Computer Science Issues*, 13(4), 52-58. https://www.ijcsi.org/papers/IJCSI-13-4-52-58.pdf
8. Liu, P.-J., Li, C.-P., & Chen, H. (2024). Enhancing storage efficiency and performance: A survey of data partitioning techniques. *Journal of Computer Science and Technology*, 39, 346-368. https://doi.org/10.1007/s11390-024-3538-1
9. Anchlia, A. (2024). Enhancing query performance through relational database indexing. *International Journal of Computer Trends and Technology (IJCTT)*, 72(8), 130-133. https://doi.org/10.14445/22312803/IJCTT-V72I8P119
10. Голубінка, В., & Худий, А. (2024). Підвищення продуктивності запитів до баз даних: аналіз технік індексації. *Вісник Національного університету «Львівська політехніка». Серія: Інформаційні системи та мережі*, (15), 65-73.
11. Azmat, H., & Huma, Z. (2025). Indexing strategies in SQL: Enhancing query efficiency and scalability. *Baltic Journal of Multidisciplinary Research*, 2(2), 130-138. https://balticpapers.com/index.php/bjmr/article/view/53
12. Robinson, E., & Anderson, J. (2025). Comparative study of adaptive indexing techniques for performance improvement in dynamic workloads. *The Journal of Innovation in Governance and Business Practices*, 1, 32–58. https://jigbp.com/index.php/jigbp/article/view/2
13. Wu, E., & Madden, S. (2011). Partitioning techniques for fine-grained indexing. In *Proceedings of the International Conference on Data Engineering (ICDE)* (pp. 1126-1137). http://hdl.handle.net/1721.1/63110

Отримано редакцією журналу / Received: 09.02.26
Прорецензовано / Revised: 25.02.26
Схвалено до друку / Accepted: 25.06.26