МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

ВАН ЧАО

ОПТИМІЗАЦІЯ ЗАПИТІВ У РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ ІЗ ВИКОРИСТАННЯМ ІНДЕКСІВ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
БУЛАТЕЦЬКА ЛЕСЯ ВІТАЛІЇВНА,
кандидат фізико-математичних наук, доцент

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних
наук та кібербезпеки
від _____ 2025 р.
Завідувач кафедри
(_____)_____
(підпис) ПІБ

ЛУЦЬК – 2025

ЗМІСТ

LIST OF ABBREVIATIONS

RDBMS – Relational Database Management System.

SQL – Structured Query Language.

CBO – Cost-Based Optimizer.

I/O – Input/Output.

PK – Primary Key.

DML – Data Manipulation Language.

MVCC – Multiversion Concurrency Control.

IOT – Index-Organized Table.

GiST – Generalized Search Tree.

GIN – Generalized Inverted Index.

ВСТУП

Сучасні інформаційні системи генерують і обробляють дуже великі обсяги транзакційних та аналітичних даних. Реляційні системи керування базами даних (СКБД) залишаються широко використовуваними, оскільки забезпечують чітку модель даних, сильну узгодженість і декларативну мову запитів. Однак у міру зростання таблиць до десятків або сотень мільйонів рядків навіть добре сформовані SQL-запити можуть демонструвати високу затримку та значне споживання ресурсів. Основна проблема полягає в зменшенні витрат на пошук, об’єднання та впорядкування релевантних даних без втрати коректності чи підтримуваності.

Індекси є основним механізмом прискорення доступу до даних у СКБД. Підтримуючи допоміжні структури над одним або кількома стовпцями, індекси дозволяють оптимізатору замінювати повне сканування вибірковими пошуками, задовольняти вимоги до сортування без додаткових операцій упорядкування та іноді повертати результати безпосередньо з індексу (покриваючого), не звертаючись до базової таблиці. Водночас індекси споживають додатковий простір і створюють навантаження під час вставки, оновлення та видалення даних. Невдало обрані або неправильно підтримувані індекси можуть уповільнювати запити чи призводити до нестабільності планів виконання. Тому важливо знати, коли індекс є корисним, який тип обрати та як узгодити його проєктування з реальними навантаженнями.

Мета дослідження. Основною метою цього дослідження є кількісна оцінка впливу різних типів і конфігурацій індексів на продуктивність запитів у реляційних базах даних, а також розроблення практичних рекомендацій щодо вибору та налаштування індексів для типових сценаріїв навантаження.

Для досягнення цієї мети передбачено виконання таких завдань:

Завдання роботи:

- проаналізувати теоретичні основи індексування в реляційних базах даних, зокрема типи індексів (B-Tree, Hash, Bitmap тощо) та їхні особливості;

- дослідити вплив різних типів індексів на продуктивність SQL-запитів, включаючи операції вибірки, фільтрації, сортування та з’єднання таблиць;
- розробити тестове середовище та модель даних для проведення експериментів із використанням індексів;
- створити набір SQL-запитів для порівняльного аналізу, що охоплюють прості, складні та високонавантажені сценарії;
- провести експериментальні вимірювання часу виконання запитів з індексами та без них при різних обсягах даних;
- оцінити вплив індексів на використання ресурсів системи — оперативної пам’яті, процесора, дискових операцій;
- виконати аналіз результатів тестування та визначити, у яких випадках застосування індексів є найбільш ефективним;
- розробити рекомендації щодо створення та використання індексів для підвищення продуктивності SQL-запитів у реляційних базах даних;
- сформувати висновки про доцільність та межі використання індексів, враховуючи потенційні недоліки (перевитрати на оновлення, збільшення обсягу даних).

Об’єктом дослідження є процес виконання SQL-запитів у реляційних системах керування базами даних за різних конфігурацій індексів і моделей навантаження.

Предметом дослідження є оптимізація запитів на основі індексів — типи та схеми побудови індексів, оцінка вартості виконання запитів на основі статистики, вибір шляхів доступу та з’єднань, а також їхній вимірюваний вплив на продуктивність (затримку, логічні/фізичні операції введення-виведення, стабільність планів).

Практичне значення полягає у формуванні прикладних, підтверджених експериментально рекомендацій щодо вибору та налаштування індексів, які зменшують затримку запитів, обсяг операцій введення-виведення та нестабільність планів виконання у реальних системах.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ЗАПИТІВ З ВИКОРИСТАННЯМ ІНДЕКСІВ

1.1 Основи реляційних баз даних та виконання SQL-запитів

В реляційних базах даних дані організовані у відношення (таблиці), що складаються з кортежів (рядків) і атрибутів (стовпців). Кожна таблиця моделює певний тип сутності, наприклад клієнтів, замовлення чи товари, тоді як обмеження забезпечують цілісність даних. Первинні ключі гарантують унікальність рядків, а зовнішні ключі посилаються на первинні ключі інших таблиць, щоб подавати зв’язки. Модель підтримує декларативне виконання запитів. SQL визначає, що потрібно отримати, а система вирішує, як зробити це ефективно [1–2].

Проєктування зазвичай починається з нормалізації. Нормальні форми (від 1НФ до БКНФ/3НФ) зменшують аномалії вставки, оновлення та видалення шляхом декомпозиції таблиць відповідно до функціональних залежностей [3]. Нормалізовані схеми мінімізують надмірність і часто створюють вузькі таблиці, які добре індексуються. Для аналітичних систем вибіркове денормалізування може зменшити кількість з’єднань і краще відобразити типові шаблони доступу. Обидва підходи впливають на фізичне виконання запитів і корисність індексів.

Фізична організація визначає, як дані зчитуються зі сховища. Сторінки (блоки) є одиницею введення/виведення. Рядки розміщуються на сторінках, що зберігаються у файлах, об’єднаних у табличні простори або фпайлові групи. Heap-таблиці зберігають рядки без певного порядку, тоді як кластеризовані або індексно-організовані таблиці впорядковують їх за ключем, що дозволяє ефективно виконувати діапазонні сканування та підвищує локальність. Рядково-орієнтоване зберігання переважає в OLTP-навантаженнях, тоді як колонарні формати використовуються в сховищах даних, оскільки прискорюють сканування та стискання при аналітичних запитах [4–5]. Такі параметри, як

структура запису, коефіцієнт заповнення, фрагментація та поділи сторінок, також впливають на поведінку введення/виведення.

Виконання SQL-запиту проходить через етапи синтаксичного розбору, переписування, оптимізації та виконання.

Розбір (parsing) розбиває SQL на токени, перевіряє синтаксис і типи, а також звіряє посилання з каталогом.

Переписування (rewriting) розгортає представлення (views), наближає предикати до базових таблиць, розгортає вкладені підзапити, спрощує вирази та усуває зайві операції. Ці трансформації зберігають семантику, але відкривають простір для ефективніших варіантів.

Оптимізація вибирає план виконання з великої множини алгебраїчно еквівалентних формулювань. Серед трансформацій: перестановка з'єднань, вибір алгоритмів з'єднання, вибір шляху доступу (послідовне сканування, індексне сканування, індексний пошук із вибіркою ключів), а також вставлення або видалення операцій сортування та матеріалізації. Пошук оптимального плану обмежується динамічним програмуванням, евристиками та відсіканням, щоб уникнути надмірної складності.

Оцінка вартості базується на моделі, що враховує витрати на I/O, CPU та пам'ять. Тому дуже важлива точна статистика: кількість рядків у таблицях та індексах, кількість різних значень, частка NULL, міри кореляції та гістограми розподілів [6]. Ці оцінки впливають на вибір між вкладеними циклами та хеш-з'єднанням, необхідність сортування і найефективніший шлях доступу. Регулярне оновлення статистики (через auto-analyze або заплановані завдання) підтримує точність, але перекошені дані, корельовані предикати та складні вирази можуть вводити оптимізатор в оману. Розширена статистика та вдало підібрані індекси допомагають пом'якшити ці проблеми.

Шляхи доступу визначають, як таблиця зчитується:

- послідовне сканування читає всі сторінки – ефективне для повного перегляду таблиці або низької вибірковості;

- індексне сканування/пошук знаходить ключі або діапазони значень, після чого зчитує базові рядки.

Покривні індекси (covering indexes) містять усі потрібні стовпці, тому звернення до базової таблиці не потрібне, що суттєво зменшує I/O.

Впорядковані структури, такі як B-дерева, ефективно обробляють діапазонні предикати [7]. Для стовпців із малою кардинальністю бітові індекси (bitmap indexes) дозволяють швидко поєднувати великі множини даних.

Основні алгоритми з'єднання [8–9]:

- вкладені цикли (Nested Loops Join) – ітерація по зовнішніх рядках і пошук відповідностей у внутрішній таблиці, ефективно при наявності індексу на ключі з'єднання;
- хеш-з'єднання (Hash Join) – створення хеш-таблиці для меншої таблиці та перевірка відповідностей із більшою; ефективно для великих не впорядкованих даних;
- злиття (Merge Join) – вимагає впорядкованості обох таблиць за ключами з'єднання; працює швидко, якщо впорядкування вже існує;

Агрегація і сортування також мають альтернативні стратегії: хеш-агрегація, сортування з подальшим об'єднанням груп або використання індексу, який вже забезпечує потрібний порядок.

Двигуни виконання (execution engines) організовують оператори у конвеєри, які виробляють і споживають кортежі.

Ітераційна модель (Volcano) отримує один рядок за раз – проста і гнучка.

Векторизовані двигуни обробляють пакети рядків, покращуючи ефективність кешу процесора [10].

Багато систем підтримують паралелізм запитів – поділ сканувань між потоками, паралельні з'єднання та одночасні агрегації.

Керування транзакціями забезпечує властивості ACID. Контроль одночасності реалізується за допомогою двофазного блокування (2PL) або багатовершинного контролю версій (MVCC). MVCC дозволяє читачам бачити

стабільну копію даних без блокування записів, що підвищує продуктивність при читанні. Застарілі версії видаляються фоново [11].

Кешування також впливає на затримку. Буферні пули зберігають часто використовувані сторінки, кеші планів – повторно використовувані плани виконання, а кеш ОС додає додатковий рівень буферизації. Тому експерименти мають контролювати стан кешу – повторювати вимірювання, очищати кеш або варіювати параметри. Інакше різниця може бути зумовлена “прогрівом” кешу, а не самим індексом [12].

Команди EXPLAIN та EXPLAIN ANALYZE дають змогу бачити обраний план: оператори, порядок з’єднань, шляхи доступу, оцінені та фактичні кількості рядків [13]. Розбіжності між оцінками й реальністю вказують на застарілу статистику чи приховану кореляцію. Для дослідження індексів ці плани підтверджують, чи використано індекс, чи є він покривним і де відбуваються сортування або “spill”-операції.

Оптимізатор використовує алгебраїчні еквівалентності та метадані, щоб розглядати семантично однакові варіанти. Індекси розширюють набір низьковитратних планів, даючи змогу виконувати пошуки, впорядковані сканування та покривні вибірки. Водночас погано спроєктовані або надлишкові індекси створюють накладні витрати на запис, збільшують розмір рядків, викликають поділи сторінок і можуть спотворювати оцінку вартості, що призводить до повільніших планів.

1.2 Основні типи індексів

Індекси – це допоміжні структури даних, які прискорюють доступ до інформації, підтримуючи впорядкування або відображення одного чи кількох стовпців. Замість того щоб переглядати кожен рядок для пошуку збігів, система переходить по індексу, щоб знайти відповідні ідентифікатори рядків (rowid, bookmark, кластерні ключі), а потім або безпосередньо повертає потрібні стовпці (якщо індекс є покриваючим), або отримує базові рядки [14]. Основний компроміс очевидний: швидше читання за рахунок більшого обсягу пам’яті та

додаткових витрат під час запису. Мета індексу – зменшити кількість операцій введення-виведення (I/O) та навантаження на процесор, звужуючи простір пошуку. Типові переваги індексів включають:

- швидкий пошук за точним значенням;
- ефективне вибіркове отримання діапазонів;
- швидші об’єднання таблиць (якщо ключі з’єднання проіндексовані);
- усунення необхідності додаткового сортування, коли індекс уже забезпечує потрібний порядок;
- можливість покриваючих запитів, коли звернення до базової таблиці не потрібне.

Ефективність індексу залежить від готовності виразу до пошуку. Умови, які мають вигляд “стовпець оператор константа”, дозволяють оптимізатору використовувати впорядковану навігацію. Якщо ж стовпець обгорнуто у функцію, звичайний індекс зазвичай не використовується, якщо тільки не створено спеціальний індекс на основі виразу (function-based index) [15].

B-tree індекси. У більшості реляційних систем за замовчуванням використовуються індекси типу B-tree або їхні варіанти (наприклад, B+-tree). Ключі зберігаються у відсортованому порядку. Внутрішні вузли спрямовують пошук, а листові вузли містять ключові значення та посилання на рядки таблиці (рис. 1.1). Часова складність пошуку, вставки й видалення є логарифмічною від кількості елементів, а структура оптимізована для блочного введення-виведення [16].

B-tree чудово підходять для умов рівності та діапазонних запитів (=, <, >, BETWEEN, пошук за префіксом у тексті з відповідним кодуванням). Оскільки листки впорядковані, сканування може повертати відсортовані результати, що дозволяє уникати додаткового сортування у виразах ORDER BY, з’єднаннях типу merge або віконних функціях.

Складені (composite) індекси типу B-tree підпорядковуються зліва направо: індекс за полями (customer_id, order_date) ефективно обслуговує умови, які обмежують customer_id і, за потреби, order_date. Проте фільтр лише за order_date

зазвичай не може скористатися цим індексом, якщо лише система не здатна отримати вигоду від впорядкованого сканування.

Для підвищення ефективності покриваючих запитів деякі СКБД дозволяють зберігати «включені» (included), тобто неключові стовпці у листових вузлах, щоб найпоширеніші запити могли читати всі потрібні дані без звернення до основної таблиці [17].

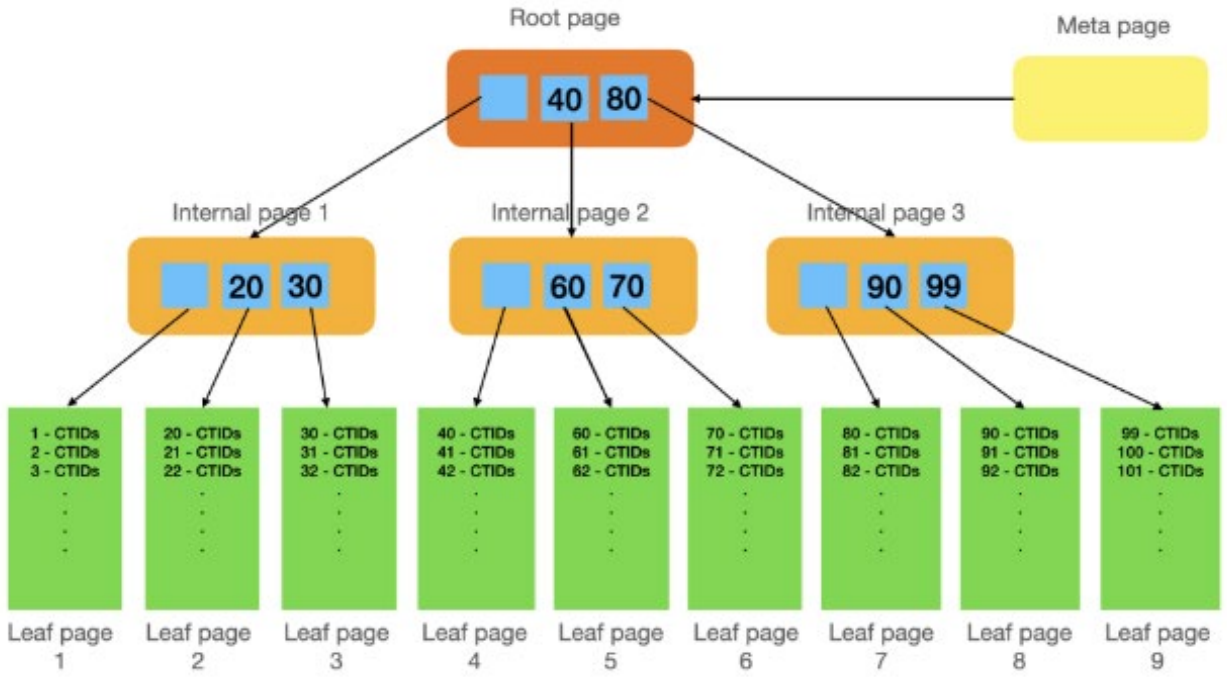

Рисунок 1.1 – Приклад індексів типу B-дерево

Хеш-індекси. Хеш-структури відображають значення ключів у кошики (buckets) за допомогою хеш-функції. Вартість пошуку очікувано становить O(1), що робить їх привабливими для запитів із чистими перевірками на рівність по високоселективних стовпцях. Однак вони не забезпечують упорядкування, тому для діапазонних сканувань, операцій ORDER BY і об’єднань типу merge join вони не приносять користі (рис. 1.2).

Характеристики оновлення залежать від системи. У деяких реалізаціях відбувається переповнення або повторне хешування під час зростання таблиць, а ступінь навантаження при записі визначається способом обробки колізій [18].

Оскільки B-дерева вже ефективно виконують пошук за рівністю та додатково підтримують діапазони й упорядкування, хеш-індекси мають обмежене застосування у багатьох робочих навантаженнях. Вони демонструють найкращі результати, коли домінують перевірки на рівність, а СУБД має зрілу й надійну реалізацію хеш-індексів із захистом від збоїв.

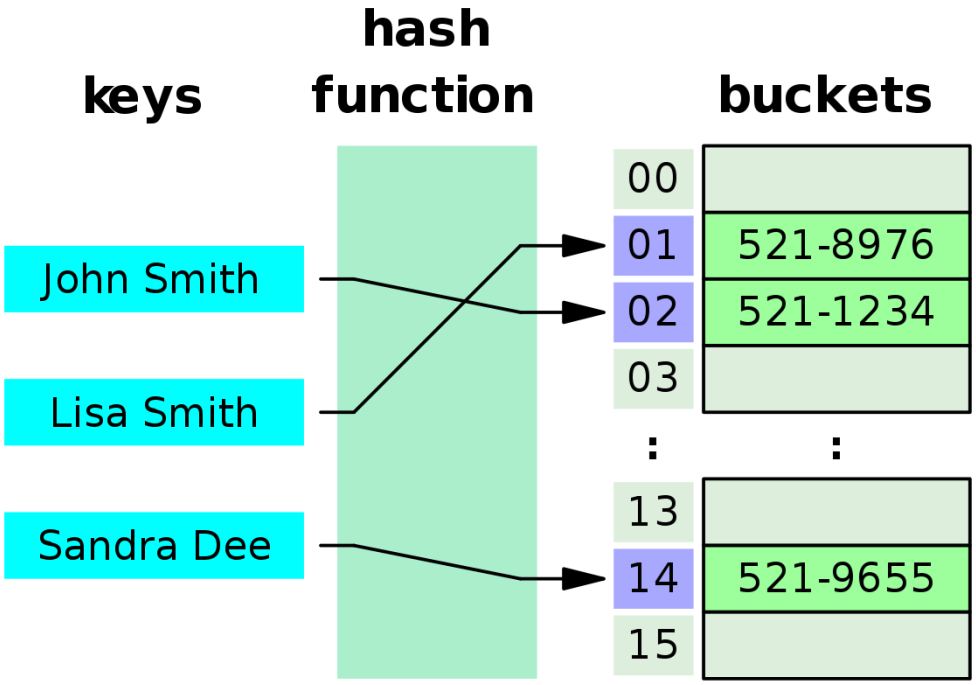

Рисунок 1.2 – Приклад хеш-індексів

Бітові індекси (Bitmap indexes). Бітові індекси представляють наявність значення за допомогою бітового вектора над ідентифікаторами рядків. Для певного значення стовпця бітова маска довжиною N (кількість рядків) показує, які рядки містять це значення. Побітові операції (AND/OR/NOT) дозволяють швидко поєднувати бітові маски, що робить бітові індекси ефективними для довільних комбінацій предикатів із низькою кількістю унікальних значень, таких як стать, статус або регіон [19]. Їх часто застосовують в аналітичних середовищах, де дані завантажуються пакетно та рідко оновлюються. Основним недоліком є вартість запису: модифікація одного рядка може зачіпати кілька великих бітових масок, тому робочі навантаження з високим рівнем паралелізму

часто страждають, рис. 1.3. Деякі системи розрізняють «бітовий індекс» (постійну структуру) та «сканування бітового індексу» (техніку виконання, що синтезує бітові маски з кількох B-деревоподібних сканувань); лише перший варіант спричиняє значні витрати на супровід операцій DML [20].

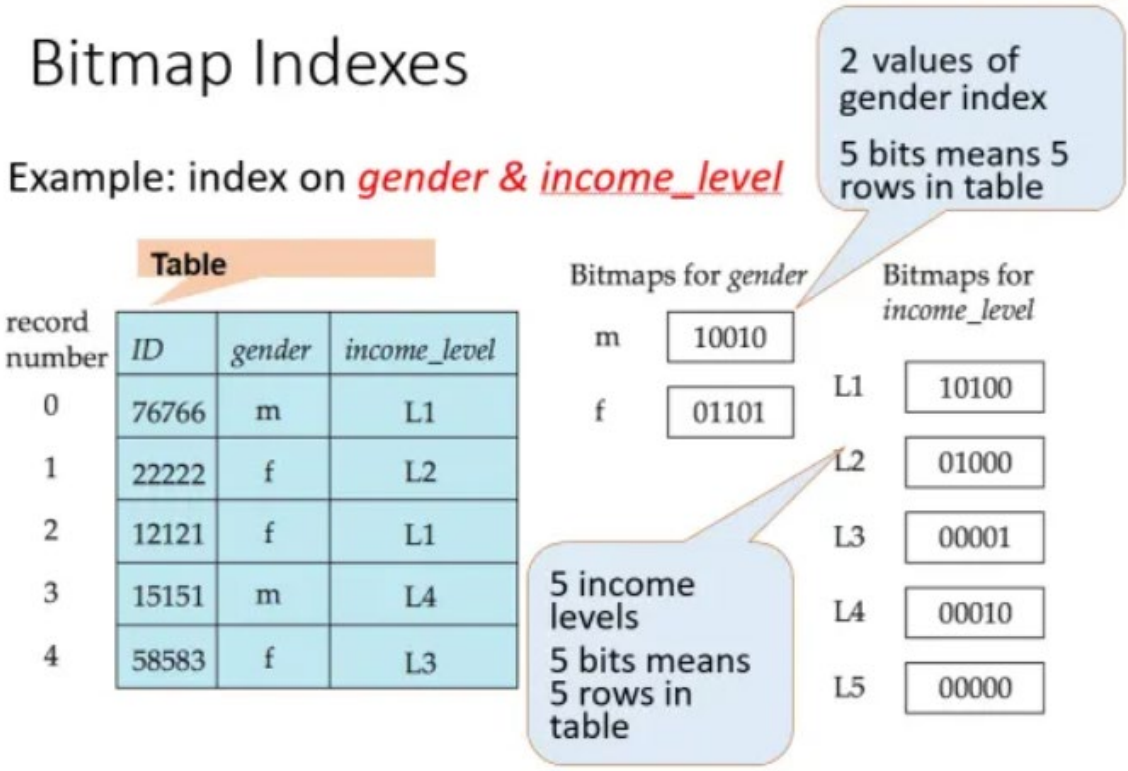

Рисунок 1.3 – Приклад бітових індексів

Кластеризоване та індексно-організоване зберігання. Кластеризований індекс (MS SQL Server) або індексно-організована таблиця (Oracle) визначає фізичний порядок рядків таблиці. Ключ кластеризації стає «хребтом» зберігання таблиці, тому діапазонні сканування за цим ключем демонструють відмінну локальність [21]. Оскільки сторінки даних впорядковані за ключем кластеризації, некластеризовані або вторинні індекси використовують цей ключ як посилання для знаходження базових рядків. Вибір ключа кластеризації має велике значення. Монотонно зростаючий ключ зменшує кількість розділень сторінок, але може створювати «гарячі» сторінки. Випадковий ключ рівномірно розподіляє вставки, але може фрагментувати структуру.Системи без явного кластеризованого індексу зазвичай все одно підтримують heap-таблиці та окремі B-дерева; деякі

підтримують функції «cluster», які розміщують пов’язані рядки разом для прискорення об’єднань [22], рис. 1.4.

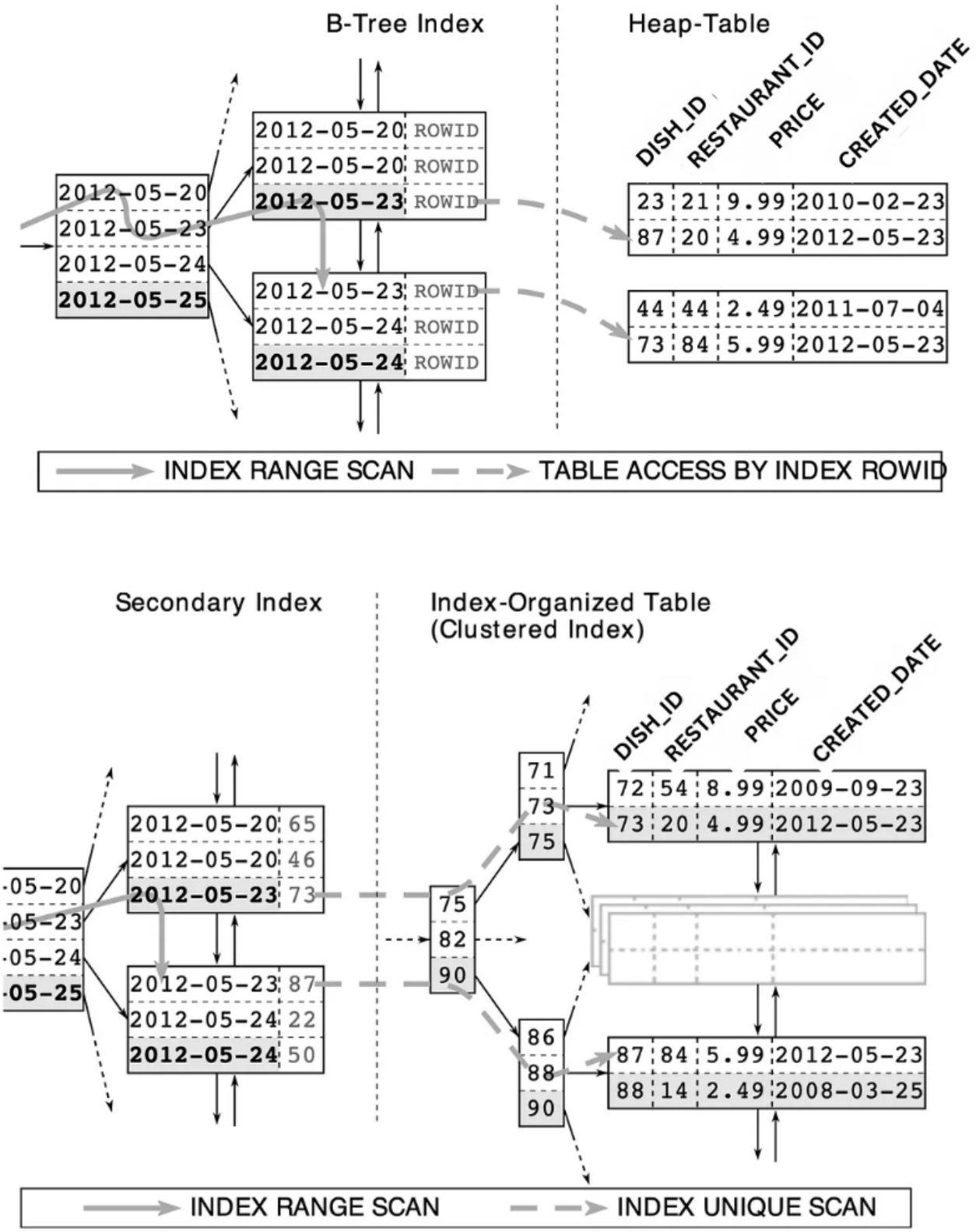

Рисунок 1.4 – Приклад кластеризованого та індексно-організованого зберігання

Складені (багатостовпцеві) індекси. Складені індекси фіксують поширені поєднання предикатів і ключів з'єднання. Їхня користь базується на узгодженості предикатів із префіксом індексу та концентрації селективності у провідних стовпцях [23]. Порядок стовпців має значення. Слід розміщувати найбільш селективний або найчастіше обмежуваний стовпець першим, якщо тільки потрібний порядок не вимагає іншого розташування. Складені індекси також можуть бути спроектовані так, щоб покривати часті запити, включаючи всі потрібні стовпці (рис. 1.5–1.6).

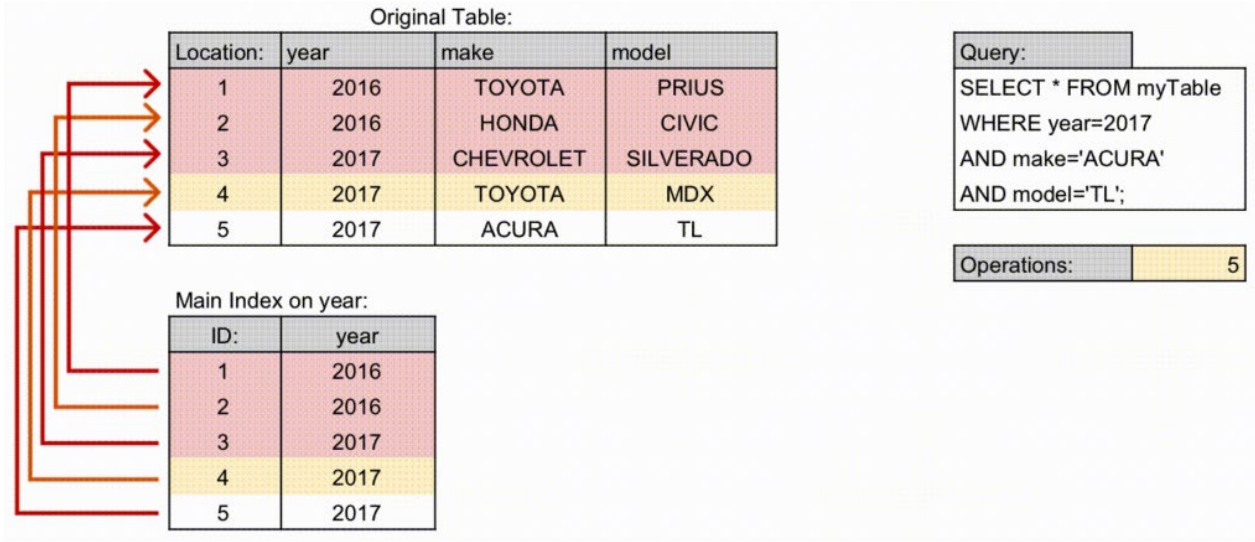

Рисунок 1.5 – Традиційний індекс

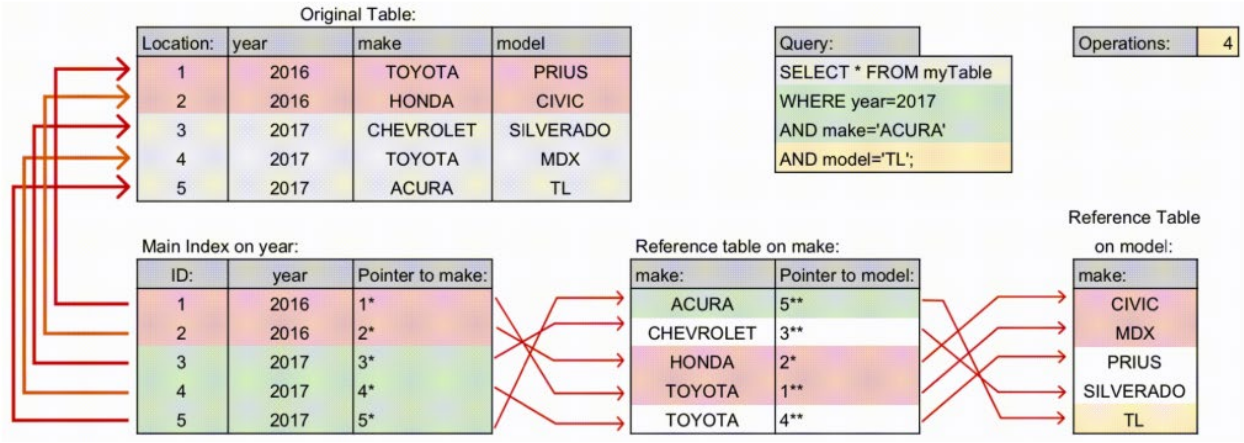

Рисунок 1.6 – Багатостовпцевий індекс

Покриваючі індекси дозволяють уникнути звернення до базової таблиці та значно зменшити затримку, але великі індекси збільшують використання пам’яті та накладні витрати при записах.

Унікальні індекси та обмеження. Унікальні індекси забезпечують, щоб значення ключів не повторювалися, що одночасно підтримує цілісність даних та оптимізацію. Кардинальність унікальна за визначенням, що гарантує високу селективність при пошуку за рівністю. Первинні ключі зазвичай реалізуються через унікальні індекси. Пошук за зовнішніми ключами виграє від індексації обох сторін зв’язку (унікальний на ключі батьків, неунікальний на стовпці-наступнику), що прискорює з’єднання та перевірку каскадних операцій [24].

Часткові та фільтровані індекси. Частковий індекс зберігає записи лише для рядків, що задовольняють предикат (наприклад, status='ACTIVE'). Коли багато запитів фільтрують за однією умовою, частковий індекс зменшує розмір та підвищує селективність, прискорюючи доступ. Накладні витрати на обслуговування стосуються лише рядків, що підпадають під умову. Коректність залежить від того, що оптимізатор розуміє предикат. Статистика по відфільтрованій підмножині допомагає моделі витрат обрати індекс у відповідних випадках [25].

Індекси на основі функцій та виразів. Коли предикати включають вирази, наприклад UPPER(name) = 'ALICE' або date_trunc('month', ts) = '2025-11-01', звичайний індекс на стовпці не є sargable. Індекс на основі виразу зберігає обчислені значення в ключі індексу, тож предикат стає прямою перевіркою на рівність або діапазон. Такі індекси ефективні для пошуку без врахування регістру, групування дат та похідних атрибутів, але повинні точно відповідати виразу, який використовується у запитах (включно з кодуванням і приведенням типів) [26].

Розширювані індексні фреймворки. Деякі системи підтримують спеціалізовані індекси:

- GiST (Generalized Search Trees) – підтримка користувацького впорядкування та семантики відстаней для геометрії, діапазонів і повнотекстового пошуку;
- SP-GiST – розбиття даних за допомогою tries або k-d дерев;

- GIN – прискорює інверсні пошуки (наприклад, масиви, лексеми тексту) [27-29].

Хоча ці методи не є центральними для транзакційних індексів, вони демонструють загальний принцип: найкращий індекс відповідає структурі даних та предикатам робочого навантаження.

Загальні принципи використання індексів. Ефективність індексу залежить від селективності та точних статистичних даних. Високоселективні предикати виправдовують використання index seek. Низькоселективні фільтри можуть виграти від комбінацій (наприклад, bitmap AND) або якщо індекс забезпечує потрібний порядок чи покриття. Гістограми, списки найбільш поширених значень та метрики кореляції допомагають оптимізатору. Застаріла або відсутня статистика може призвести до помилкової оцінки витрат і пропуску корисного індексу.

Кожен додатковий індекс створює накладні витрати на INSERT, UPDATE та DELETE:

- вставки додають ключ у всі відповідні індекси;
- оновлення індексованих стовпців вимагає видалення та вставки в дереві індексу;
- видалення видаляє записи.

Розділення сторінок, фрагментація на рівні листів і ребалансування споживають I/O та можуть збільшувати затримку запису. Налаштування fill factor, періодична реорганізація та уважний вибір індексованих стовпців зменшують ці ефекти [30]. Для таблиць з великою кількістю записів найчастіше мінімальна, але цільова індексація працює ефективніше, ніж широка спекулятивна.

Практична ефективність також залежить від деталей:

- кодування визначає порядок рядків і чутливість до регістру;
- деякі системи трактують NULL як високі значення, інші дозволяють IS NULL використовувати індекс;
- неявне приведення типів може відключити використання індексу.

Форма запису предикатів (sargable), уникнення функцій по індексованій стороні, узгодження типів даних та використання діапазонів відповідно до порядку індексу часто дає більший ефект, ніж додавання нових індексів.

Отже, B-tree – універсальний інструмент для OLTP і багатьох аналітичних запитів; Hash-індекси – підходять для чистих перевірок на рівність по висококардинальних ключах, якщо реалізація надійна; Bitmap-індекси – для аналітики з рідкими оновленнями, низькоселективними фільтрами та складними комбінаціями; Складені, часткові та функціональні індекси адаптують індекси під реальні навантаження, концентрують селективність, забезпечують покриття і відновлюють sargability. Оптимальний набір індексів визначається навантаженням. Потрібно вимірювати частоту та селективність предикатів, проектувати мінімальні індекси для найпоширеніших запитів і з’єднань, і перевіряти вибір за допомогою EXPLAIN та метрик виконання.

1.3 Статистика, селективність та кардинальність

Оптимізація запитів на основі вартості (Cost-based query optimization) спирається на кількісну характеристику даних. Цю характеристику забезпечують статистичні дані, які оптимізатор використовує для оцінки кардинальності (кількості рядків, що проходять через оператори плану) та селективності (частки рядків, що задовольняють предикати). Коли ці оцінки близькі до реальності, оптимізатор може обирати ефективні шляхи доступу, порядок з’єднань та реалізацію операторів. Якщо оцінки неправильні, плани можуть зайвий раз читати дані, записувати на диск або перевантажувати кеші.

На рівні стовпців бази даних зберігають компактні підсумки: загальну кількість рядків, кількість унікальних значень, частку NULL та гістограми, що апроксимують розподіл значень. Гістограми однакової висоти (equ- height) поширені, оскільки кожен бакет містить подібну кількість рядків, що забезпечує надійну оцінку діапазонної селективності навіть при перекосі даних. Списки найбільш поширених значень (most-common-values) доповнюють гістограми, відображаючи точну частоту для «важких» значень, які могли б спотворити уявлення про рівномірність. Ці підсумки навмисно зберігаються компактними для зменшення витрат на обслуговування, але достатньо виразними, щоб уникнути наївних оцінок, наприклад «рівність = 1/NDV» [31].

Моделювання кардинальності ускладнюється, коли запити накладають обмеження на кілька стовпців одночасно. Множення незалежних селективностей рідко відповідає реальності, оскільки атрибути корелюють: місто передбачає невеликий набір поштових індексів, категорія продукту корелює з типовими ціновими діапазонами, статусні прапорці групуються за етапами життєвого циклу. Щоб враховувати такі випадки, використовують розширену статистику, що включає багатостовпцеві підрахунки унікальних значень, функціональні залежності або спільні гістограми. Завдяки цьому фільтр по одному атрибуту може інформувати про розподіл іншого, стабілізуючи оцінки як для предикатів на одній таблиці, так і для подальших з’єднань.

Оцінка з’єднань залежить від перетину доменів значень. Для рівноставних з’єднань (equality joins) на незалежних ключах добре відома евристика обмежує розмір результату як добуток входів, поділений на більшу кількість унікальних значень у ключових стовпцях. Це обмеження уточнюють ситуацію: первинні ключі встановлюють верхню межу розгалуження, зовнішні ключі не дозволяють розширення поза межі посилальної сторони, а перевірки й партиційні предикати звужують ефективні діапазони. При сильному перекосі розподілу критичними стають гістограми та списки найбільш поширених значень. Без них оптимізатор може обрати план nested loops із мільйонами звернень або, навпаки, виділити занадто великий хеш, що записується на диск.

Форма предикату безпосередньо впливає на застосування статистики. Вирази, такі як date_trunc(ts) або UPPER(name), порушують простий шаблон «column op constant». Якщо система підтримує статистику для обчислених виразів або збережених обчислених стовпців, оцінка використовує ці розподіли. Інакше застосовуються консервативні значення за замовчуванням, що часто призводить до оборонних планів із додатковими сортуваннями чи хеш-таблицями. Узгодження форм предикатів зі статистикою (уникнення непотрібних функцій по індексованій стороні або введення статистики для виразів) відновлює передбачуваність.

Статистика є надійною лише настільки, наскільки свіжою є інформація. Автоматичне обслуговування зазвичай спрацьовує при зміні певної частки рядків, але робочі навантаження з частими невеликими оновленнями можуть накопичувати дрейф. Масові завантаження, масові видалення або перерозподіл – природні моменти для явного оновлення статистики. Таблиці з партиціями виграють від локальної статистики (для кожної партиції) та глобальних підсумків: локальна статистика зберігає сезонність та регіональні перекоси, глобальна – забезпечує коректну оцінку крос-партиційних запитів. Також важлива стратегія вибірки. Надто малі вибірки пропускають рідкісні, але важливі значення. Надто великі – збільшують накладні витрати без суттєвої користі. Орієнтовані гістограми з високою деталізацією для критичних стовпців предикатів часто приносять найкращий результат.

Зворотний зв’язок під час виконання (execution feedback) закриває цикл між оцінками та реальністю. Деякі системи записують спостережувані кардинальності для конкретних предикатів та коригують майбутні оцінки, інші адаптуються під час виконання, перепартиціонуючи hash joins або змінюючи стратегії, коли лічильники перевищують пороги. Ці механізми зменшують залишкову похибку оцінки, особливо при складних кореляціях або нестабільних даних, але вони залежать від повторюваних навантажень і можуть не допомагати разовим аналітичним запитам [32].

Кешування планів та параметризація додають ще одну складність. План, скомпільований під високоселективне літеральне значення, може використовуватися для низькоселективного параметру і навпаки. Така невідповідність, часто звана parameter sniffing, пояснює, чому план index-seek з ключовими зверненнями добре працює для одного значення і погано для іншого. Рішення залежать від платформи: компіляція «для невідомого», використання план-гайдів або підказок, включення перегенерації плану під час виконання для змінних запитів. Головна ідея з точки зору статистики — уникати оцінки селективності за нетиповими вибірками.

Для експериментальної оцінки статистика визначає методику та інтерпретацію результатів. Перед вимірюванням статистика має бути актуальною, щоб уникнути коливань планів через застарілі метадані. Під час вимірювання слід фіксувати вивід EXPLAIN або EXPLAIN ANALYZE разом із часом виконання, щоб порівняти оцінені та фактичні кількості рядків на ключових операторах. Постійні розбіжності вказують на відсутні гістограми, пропущені залежності або невідповідну форму предикату [33]. Для складних предикатів корисно зазначати наявність розширеної статистики. Її відсутність пояснює, чому очевидне поєднання не створює індекс-дружній план. У партиційованих наборах даних повторення тестів на різних партиціях допомагає оцінити, чи локальна статистика відображає зміни розподілу, що впливають на вибір плану.

Операційно, добре налаштована статистика зменшує потребу у довільних підказках та забезпечує стабільну продуктивність при оновленні даних. Вона:

- керує виділенням пам’яті для хеш-таблиць та сортувань, запобігаючи spills;
- дозволяє оптимізатору розпізнавати, коли потрібний порядок вже існує, уникаючи непотрібних сортувань;
- робить порядок з’єднань стійким до рутинних змін обсягів даних.

Натомість нехтування статистикою може зробити індексацію неефективною. Підходящий індекс може бути проігнорований, якщо предикат

неправильно оцінено як неселективний, тоді як непотрібний індекс може здаватися «корисним» лише через неправильну оцінку, що змусила оптимізатор використати його для мінімізації шкоди, а не для підвищення ефективності.

1.4 Вплив індексів на продуктивність запитів

Індекси змінюють спосіб, у який база даних знаходить, впорядковує та об’єднує дані, впливаючи на обсяг операцій вводу-виводу I/O, навантаження на CPU, використання пам’яті та форму плану виконання. Їхні переваги залежать від селективності предикатів, необхідного порядку, охоплення стовпців та взаємодії з операторами об’єднання й агрегування. Неправильно підібрані індекси можуть бути проігноровані або, що гірше, збільшити загальне навантаження через додаткові звернення та обслуговування.

Без індексу запит зазвичай виконує послідовне сканування, торкаючись більшості сторінок даних. Відповідний індекс дозволяє виконувати пошук і короткі сканування діапазону, які рано відсікають непотрібні сторінки. У row-oriented сховищах пошук за індексом підтримує випадкові I/O операції, послідовне сканування – великі суміжні читання. На системах із SSD випадковий I/O дешевший, ніж на HDD, але все ще дорожчий за послідовне потокове читання на великих обсягах. Якщо індекс не охоплює всі необхідні стовпці, кожен відповідний ключ може викликати пошук (bookmark/rowid) у базовій таблиці. Співвідношення кількості таких звернень до корисних рядків визначає, чи вигідний шлях через індекс у порівнянні зі скануванням.

Охоплюючий (covering) індекс надає всі стовпці, необхідні для запиту, усуваючи читання базової таблиці та значно зменшуючи затримку. Багато СУБД дозволяють включати “неключові” стовпці в листові сторінки, щоб досягти охоплення без збільшення порядку сортування. Охоплення особливо ефективне для селективних предикатів, невеликих проєкцій та точкових OLTP-запитів. Для широких проєкцій або сильно змінних предикатів охоплення може збільшити розмір індексу та погіршити розташування в кеші, компенсуючи вигоди.

Оскільки листові вузли B-дерева впорядковані, індекс, що відповідає ORDER BY або клаузі розділення/порядку віконної функції, може усунути явне сортування. Це критично для запитів з LIMIT/Top-N: індекс, впорядкований за ORDER BY created_at DESC, повертає перші K рядків із мінімальним I/O, тоді як сканування вимагало б сортування або матеріалізації значно більшого обсягу даних. Композитні індекси можуть задовольнити складні впорядкування, якщо запитуваний порядок є ліво-правим префіксом ключів індексу, а напрями збігаються або підтримуються зворотними скануваннями СУБД.

Корисність індексів зростає із селективністю. Фільтри рівності на стовпцях з високим NDV і вузькі предикати діапазону зазвичай отримують користь. Низькоселективні предикати (наприклад, status='ACTIVE', коли 80% рядків задовольняють умову) часто віддають перевагу скануванням, якщо вони не комбінуються з іншими фільтрами (bitmap AND/перетин) або коли індекс забезпечує потрібне впорядкування чи охоплення. Поріг вигідності змінюється залежно від апаратного забезпечення (SSD vs. HDD), розміру сторінки та "теплоти" кешу.

Порядок стовпців у композитному індексі визначає як пошук, так і впорядкування. Предикат на (customer_id, order_date) виграє від індексу, ключованого за (customer_id, order_date); зміна порядку зазвичай не дозволяє пошук за customer_id+order_date, якщо оптимізатор не може використати часткове впорядкування. Провідні стовпці повинні відображати найселективніші або найчастіше обмежувані предикати, якщо тільки необхідний порядок (ORDER BY date) не диктує інше розташування. Для змішаних умов, ідеально підходить композитний індекс.

Nested loops з індексними пошуками ефективні, коли зовнішнє вхідне джерело мале або високоселективне, а внутрішня сторона має індекс за ключем об'єднання. Кожен зовнішній рядок викликає дешевий пошук замість сканування. При недооцінці кардинальності зовнішньої сторони цей план може призвести до мільйонів випадкових звернень. Точна статистика робить цей шлях безпечним.

Merge joins вигідні, коли обидві сторони впорядковані за ключами об’єднання, часто через відповідні індекси, усуваючи сортування та дозволяючи потокову оцінку.

Hash joins не потребують індексів для коректності, але індекси можуть допомогти, зменшуючи вхідні дані для побудови/пошуку через попереднє фільтрування предикатів або забезпечуючи необхідне впорядкування після об’єднання, щоб уникнути сортування.

Впорядкування індексів зменшує роботу для GROUP BY/DISTINCT: сканування індексу, впорядкованого за груповими ключами, дозволяє агрегувати в порядку, пропускаючи явне сортування. Hash-агрегація ігнорує впорядкування, але виграє, коли індекси попередньо фільтрують вхідні дані. Для rollup за часом або категорією, індекси на виразах чи часткові індекси (наприклад, date_trunc('month', ts)) можуть зменшити обсяг вхідних даних і стабілізувати затримку.

Багато віконних запитів потребують конкретного розділу/порядку. Індекс, що відповідає (partition_key, order_key), може уникнути глобального сортування та дозволити обмежену оцінку кадру в порядку. Якщо тільки порядок збігається, часткова вигода зберігається. СУБД все ще може потребувати повторного розподілу, але пропускає основне сортування.

Деякі СУБД можуть поєднувати кілька одно-стовпцевих індексів, щоб задовольнити кон’юнктивні (перетин) або диз’юнктивні (об’єднання) предикати. Фізична реалізація варіюється: логічне об’єднання bitmap (сканування bitmap-індексів) або злиття списків rowid. Перетин допомагає, коли немає єдиного композитного індексу, але є кілька селективних фільтрів. Це додає накладні витрати на поєднання кандидатів і все ще може вимагати пошуку базових рядків добре підібраний композитний індекс часто перевершує перетин у повторюваних навантаженнях.

Пагінація на основі OFFSET (OFFSET N LIMIT K) сповільнюється при великих зміщеннях, оскільки СУБД доводиться читати та відкидати N рядків. Пагінація на основі ключів (keyset) із селективним, впорядкованим індексом

(WHERE (key < last_key) ORDER BY key DESC LIMIT K) масштабуються краще, читаючи точно наступні K рядків. Проєктування індексів для доступу по ключах є поширеною OLTP-оптимізацією.

Індекси прискорюють предикати тільки тоді, коли предикат може використати порядок індексу. Функції на індексованому стовпці (UPPER(name)) або невідповідність типів змушують виконувати сканування, якщо не існує індексу на вираз, який матеріалізує результат функції. Предикати діапазону повинні відповідати сортуванню та типу даних індексу; неявне приведення типів може відключити пошук і збільшити навантаження на CPU.

З кластеризованим індексом або таблицею, організованою за індексом, ключ кластеризації визначає фізичний порядок рядків. Сканування діапазону за цим ключем забезпечує відмінну локальність і менше випадкових читань. Вторинні індекси на кластеризованих таблицях використовують ключ кластеризації як bookmark, що може зменшити вартість пошуку порівняно з heap + прихований rowid. Вибір ключа кластеризації з гарною тимчасовою або просторовою локальністю (наприклад, (customer_id, order_date)) може покращити як точкові, так і діапазонні запити, але може концентрувати “гарячі” вставки.

Паралельні сканування зменшують реальний час виконання на великих обсягах. Пошук за індексом менш ефективно паралелізується, коли зовнішній драйвер малий. Hash joins і сортування потребують пам’яті. Коли оцінки недооцінюють кількість рядків (проблема статистики), оператори можуть виливатися на диск, зводячи нанівець переваги індексації. Індекси, що попередньо фільтрують вхідні дані, дозволяють утримувати операції в межах пам’яті та зменшують ризик spill.

Кожен додатковий індекс збільшує вартість INSERT/UPDATE/DELETE через обслуговування додаткових структур і може посилювати розділення сторінок та фрагментацію, що впливає на продуктивність читання через погіршену локальність. Для таблиць з великою кількістю записів вузькі, орієнтовані на навантаження індекси зазвичай перевершують широкі охоплюючі

конструкції. Потрібне періодичне обслуговування (repack/rebuild/reorg) для відновлення ефективності читання після тривалого навантаження.

Присутність індексу розширює простір пошуку плану. За наявності надійної статистики оптимізатори стабільно обирають низьковартісні плани, що використовують індекси для селективних фільтрів, необхідного порядку та охоплення. За змінних розподілів або параметрично-чутливих навантажень той самий індекс може використовуватися в одних виконаннях і ігноруватися в інших, створюючи змінну затримку. Техніки стабілізації включають розширену статистику, фільтровані індекси, що концентрують селективність, та уважну форму предикатів.

Навіть якщо фільтр допускає багато рядків, індекс може виграти, коли він забезпечує потрібний порядок, дозволяє merge join або охоплює всі необхідні стовпці з маленькими листовими сторінками. Навпаки, на перший погляд селективний індекс може програти, якщо кожне відповідне значення потребує випадкових пошуків ключів у широкій базовій таблиці.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА ЕКСПЕРИМЕНТУ ДЛЯ ОЦІНКИ ВПЛИВУ ІНДЕКСІВ НА ПРОДУКТИВНІСТЬ SQL-ЗАПИТІВ

2.1 Постановка задачі

У сучасних інформаційних системах обробка великих обсягів даних є критично важливою складовою ефективного функціонування програмних продуктів. Швидкість виконання SQL-запитів безпосередньо впливає на продуктивність застосунків, час відгуку сервісів, а також на можливість масштабування системи. Одним із ключових інструментів оптимізації запитів у реляційних базах даних є використання індексів, які дозволяють значно прискорити операції пошуку, фільтрації, сортування та з’єднання таблиць.

Попри очевидні переваги, індекси мають і певні недоліки: вони займають додатковий простір у пам’яті, уповільнюють операції вставки та оновлення, а неправильний вибір типу індексу або поля для індексування може не тільки не покращити, а й погіршити продуктивність. Тому постає задача дослідити особливості роботи різних типів індексів та визначити, в яких саме випадках їх застосування є найбільш ефективним.

У межах даної роботи необхідно сформувати модель даних, створити набір SQL-запитів різної складності та провести експериментальні вимірювання часу їх виконання з індексами та без них. На основі отриманих результатів потрібно здійснити аналіз впливу індексів на продуктивність та виробити практичні рекомендації щодо їх використання у реальних системах.

Таким чином, основною задачею дослідження є обґрунтування та експериментальна перевірка ефективності застосування індексів як методу оптимізації SQL-запитів у реляційних базах даних, а також визначення умов, за яких індексування забезпечує максимальне підвищення продуктивності.

2.2 Налаштування середовища, проєктування моделі даних та генерація набору даних

Перед початком тестування продуктивності необхідно було створити контрольоване середовище, де кожна зміна в поведінці запиту могла б бути однозначно віднесена виключно до індексування. Підготовка включала конфігурування сервера бази даних, проєктування логічної схеми, генерацію синтетичних, але реалістичних даних та перевірку їхньої цілісності.

Усі експерименти проводились на виділеній робочій станції для усунення впливу зовнішнього навантаження. Конфігурація була наступною: Intel® Core™ i7-12700 (20 потоків, базова частота 2.10 ГГц), 32 ГБ оперативної пам’яті DDR4 RAM, 1 ТБ NVMe SSD (Samsung 970 EVO Plus) та Ubuntu Server 22.04.4 LTS з ядром 5.15.0-117. Регулятор частоти процесора (CPU frequency governor) було зафіксовано на режимі "performance" (продуктивність).

Рушієм бази даних було обрано PostgreSQL за його стабільність та детальні інструменти аналізу запитів. Кластер було ініціалізовано з параметрами сортування C collation та типом символів C ctype для забезпечення детермінованого упорядкування. Кілька параметрів було налаштовано відповідно до характеристик SSD та для гарантування послідовних вимірювань: shared_buffers = 8 GB, work_mem = 128 MB, effective_cache_size = 24 GB, random_page_cost = 1.1, seq_page_cost = 1.0 та track_io_timing = on.

Під час завантаження даних автоочищення (autovacuum) було тимчасово вимкнено, а пізніше знову увімкнено для всіх тестів.

Використовувалися два режими тестування: холодні запуски (cold runs), де кеші очищалися (sync; echo 3 > /proc/sys/vm/drop_caches), та теплі запуски (warm runs), де кожен запит виконувався один раз для попереднього заповнення кешу та ще три рази для вимірювання часу. Записувалися медіанні значення.

Експериментальна схема представляє собою спрощену систему електронної комерції, розроблену для демонстрації типових реляційних

операцій: вибірки, об’єднання (joins) та агрегації. Структура містить чотири нормалізовані таблиці з первинними та зовнішніми ключами.

```
CREATE TABLE customers (
  id    BIGSERIAL PRIMARY KEY,
  name  TEXT NOT NULL,
  city  TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL
);

CREATE TABLE products (
  id       BIGSERIAL PRIMARY KEY,
  name     TEXT NOT NULL,
  category TEXT NOT NULL,
  price    NUMERIC(10,2) NOT NULL CHECK (price >= 0)
);

CREATE TABLE orders (
  id          BIGSERIAL PRIMARY KEY,
  customer_id BIGINT NOT NULL REFERENCES customers(id),
  date        TIMESTAMP NOT NULL,
  amount      NUMERIC(12,2) NOT NULL CHECK (amount >= 0)
);

CREATE TABLE order_details (
  order_id   BIGINT NOT NULL REFERENCES orders(id),
  product_id BIGINT NOT NULL REFERENCES products(id),
  quantity   INT NOT NULL CHECK (quantity > 0),
  PRIMARY KEY (order_id, product_id)
);
```

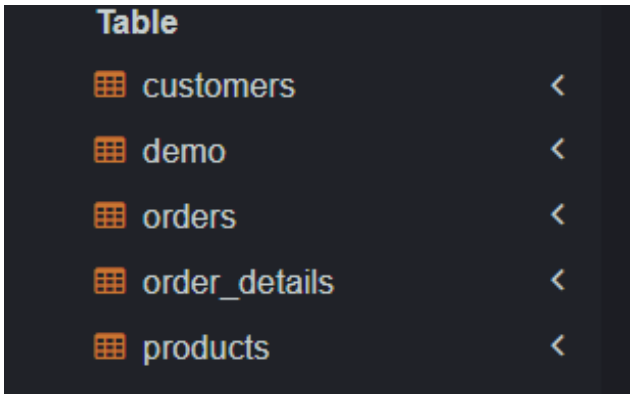

Рисунок 2.1 – Таблиці для експериментів

Для візуалізації структури на рис. 2.2 подано логічну схему. Вона демонструє всі чотири таблиці, їхні атрибути та залежності за зовнішніми ключами.

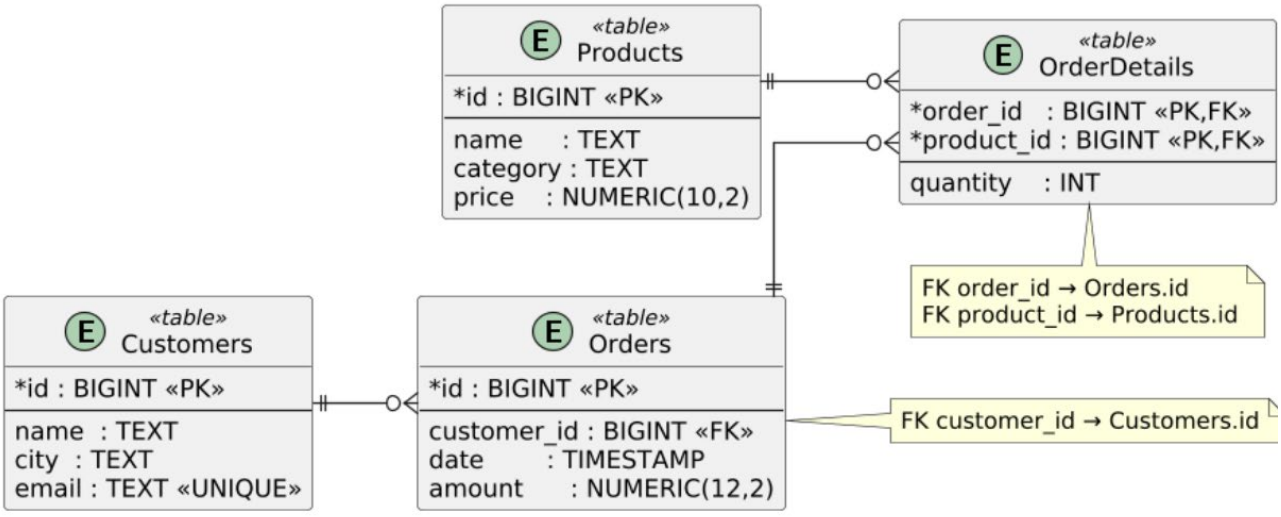

Рисунок 2.2 – Логічна схема тестової таблиці (ERD)

Ця діаграма, Рис. 2.2, чітко демонструє, що кожен покупець може мати кілька замовлень, кожне замовлення може містити кілька товарів, і кожен товар може фігурувати у багатьох замовленнях. Таблиця OrderDetails реалізує відношення «багато-до-багатьох» (many-to-many) між таблицями Orders та Products.

Після визначення схеми дані були згенеровані програмно за допомогою Python 3.11 та бібліотеки Faker 25.0.0 з фіксованим початковим значенням для генератора випадкових чисел (random seed) 20251113 (Додаток А).

Використовувалися такі обсяги даних:

- таблиця Customers (Покупці) – 50 000 рядків;
- таблиця Products (Товари) – 10 000 рядків у 12 категоріях;
- таблиця Orders (Замовлення) – 700 000 записів із датами в діапазоні між 2022-01-01 та 2025-10-31 (UTC);
- таблиця OrderDetails (ДеталіЗамовлення) – 2 000 000 записів, що пов’язують замовлення та товари.

Розподіли були розроблені таким чином, щоб нагадувати реалістичні бізнес-дані. Назви міст обиралися зі списку, що містить 200 елементів, із застосуванням розподілу Зіпфа (Zipf distribution) [36]. Категорії товарів відповідали фіксованому процентному вектору. Ціни генерувалися з логнормального розподілу ($\mu = 3.6$, $\sigma = 0.75$). Кількість позицій у замовленні відповідала трикутному розподілу (2-7, mode = 3).

Перевірки зовнішніх ключів (foreign-key checks) не виявили порушень. Діапазон дат у полі orders.date відповідав заданим межам, а середня кількість позицій у замовленні становила 2.86.

2.3 Визначення робочого навантаження та методологія вимірювання

Рішення про використання стратегії B-tree індексування було зумовлене її універсальною застосовністю та ефективністю в обробці широкого спектру типів запитів, включаючи умови рівності та діапазону. Оскільки планувальник запитів PostgreSQL автоматично адаптується до наявності індексів, було важливо протестувати кожен запит двічі: один раз на необробленій схемі (без індексів) і один раз після створення відповідних індексів. Обрані обсяги даних (50 000 покупців, 10 000 товарів, 700 000 замовлень та 2 000 000 записів деталей замовлень) гарантували, що повне сканування таблиць було достатньо

витратним для демонстрації різниці в продуктивності. Усі тести виконувались на фіксованій конфігурації PostgreSQL з ідентичними системними параметрами, і кожен запит запускався тричі для зменшення дисперсії.

У Таблиці 2.1 представлено кількісні результати аналізу робочого навантаження.

Таблиця 2.1

Середня продуктивність запитів до та після застосування індексів

№	Тип запиту	Умова	Час виконання (мс)		Приріст (×)	План виконання
			Без індексу	З індексом		
1	Пошук за рівністю	email = 'user5000@example.com'	1425	3.2	445×	Сканування індексу (idx_customer_email)
2	Запит по діапазону	date BETWEEN '2023-01-01' AND '2023-01-31'	5860	127	46×	Сканування діапазону індексу (idx_order_date)
3	З'єднання за зовнішнім ключем	orders.customer_id = customers.id	9020	1320	6.8×	Вкладений цикл + Пошук індексу
4	Групування за категорією	GROUP BY products.category	4870	2280	2.1×	Сканування купи бітмап + Хеш-агрегація
5	Сортування / розбиття на сторінки	ORDER BY date DESC LIMIT 50	2740	18	152×	Сканування індексу (idx_order_date_desc)

Кожен запит відображає окремий шаблон доступу, який часто спостерігається в корпоративних системах, від простих операцій пошуку до складних аналітичних об'єднань (joins). Час виконання вимірювався за допомогою команди EXPLAIN (ANALYZE, BUFFERS) та усереднювався за три

послідовні запуски. Базовий сценарій (без індексів) порівнювався з оптимізованою схемою, що містить B-tree індекси на ключових атрибутах, таких як електронна пошта, дата та колонки зовнішніх ключів.

До додавання індексів кожен запит вимагав повного послідовного сканування цільової таблиці, що призводило до значних операцій вводу/виводу та тривалого часу виконання. Після введення індексів планувальник запитів перейшов до ефективніших стратегій, як видно з планів EXPLAIN. Найбільшу користь отримали пошуки на основі рівності, досягнувши прискорення понад чотириста разів завдяки прямому доступу до індексів. Діапазонні запити, які покладаються на впорядкований прохід ключів, також продемонстрували значне покращення, скоротивши час обробки з кількох секунд до мілісекунд. Навіть операції агрегації та об’єднання, які зазвичай обмежені ресурсами процесора, продемонстрували помітне підвищення продуктивності, оскільки індекси дозволили PostgreSQL мінімізувати кількість сторінок диска, що зчитуються під час етапів групування та об'єднання.

Щоб краще проілюструвати ефективність індексів B-дерева, на рис. 2.3 представлено спрощену версію внутрішньої структури індексу B-дерева, створеного на основі стовпця Orders(date). Кожен рівень структури сприяє швидшій навігації по набору даних. Кореневий вузол зберігає межі ключів, спрямовуючи пошук до проміжних нелистових вузлів, які, у свою чергу, вказують на листові вузли, що містять фактичні ключі та вказівники рядків (ROWID).

Структура B-дерева, зображена на рис. 2.3, відображає ієрархію, знайдену в індексних файлах PostgreSQL. Під час виконання запиту механізм бази даних не сканує таблицю безпосередньо, а проходить це дерево від кореневого вузла до певної кінцевої сторінки, дотримуючись меж ключів. Після того, як знайдено правильний кінцевий вузол, PostgreSQL отримує лише необхідні рядки, використовуючи їхні фізичні адреси.

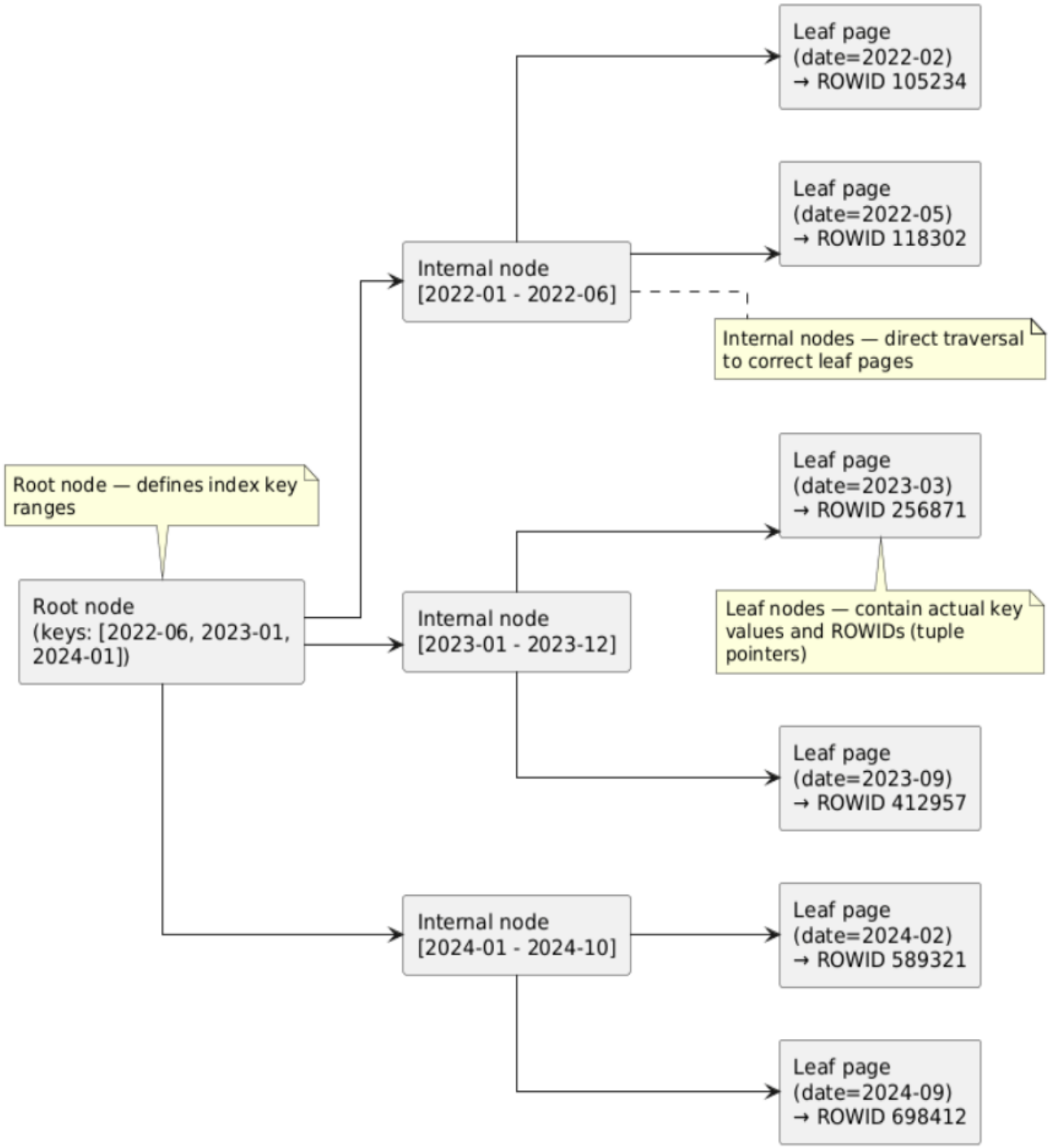

Рисунок 2.3 – Внутрішня структура індексу B-дерева, створеного на основі стовпця Orders(date)

Це значно зменшує обсяг дискового вводу-виводу, особливо для великих наборів даних, де повне сканування вимагало б тисяч зчитування сторінок. Глибина дерева залежить від розміру таблиці та кардинальності ключа, але

навіть для мільйонів записів вона рідко перевищує три або чотири рівні, а це означає, що кожен пошук вимагає лише кількох операцій вводу-виводу. Цей механізм пояснює високу ефективність, що спостерігається в експериментах, і підтверджує теоретичні характеристики продуктивності індексів B-дерева.

Проведений експеримент демонструє, що індексування – це не просто теоретична оптимізація, а практичний механізм, здатний трансформувати продуктивність запитів у великомасштабних системах. Завдяки впровадженню цільових індексів час виконання скоротився з секунд до мілісекунд, що призвело до вимірних переваг для всіх типів запитів. Результати показують, що індекси B-дерева є особливо вигідними для вибіркового пошуку, фільтрів діапазону та запитів, що включають сортування або впорядкування за ключовими атрибутами. Хоча приріст продуктивності для операцій агрегації та об'єднання був меншим, він все ще був достатньо значним, щоб виправдати індексацію стовпців зовнішнього ключа та групування атрибутів.

Візуалізована структура B-дерева додатково підтверджує ці висновки, показуючи, як PostgreSQL організовує індексовані дані в ефективну ієрархічну структуру, яка мінімізує випадковий доступ до диска. Кореневий та проміжний вузли дозволяють логарифмічну складність пошуку, тоді як кінцеві вузли забезпечують пряме відображення на рівень фізичного сховища. Отже, система досягає високої узгодженості продуктивності навіть з мільйонами записів.

2.4 Реалізація сценаріїв

Цей етап дослідження був присвячений реалізації експериментів, розроблених для вимірювання практичної ефективності різних типів індексів у PostgreSQL. Експерименти були організовані за двома основними конфігураціями: базовий сценарій, який містив лише первинні та зовнішні ключі, та оптимізована конфігурація, яка включала B-дерево, композитні та функціональні індекси. Крім того, вибрані запити були протестовані в умовах низької вибірковості, щоб продемонструвати поведінку планів виконання.

Метою було відтворити реальні робочі навантаження, в яких база даних виконує різноманітні операції пошуку, з’єднання та агрегації, а також визначити, як дизайн індексу змінює фактичні плани запитів та продуктивність.

Усі запити в базовій конфігурації виконувалися з використанням повного послідовного сканування, що призводило до високих витрат на ввід-вивід та значного часу виконання. Цей етап забезпечив контрольні вимірювання для порівняння продуктивності. PostgreSQL значною мірою покладався на оператори Seq Scan, Hash Join та Hash Aggregate, що вимагало повного читання таблиці майже для всіх робочих навантажень. Навіть вибіркові фільтри, такі як WHERE email = 'user5000@example.com', вимагали сканування всіх 50 000 рядків клієнтів.

Оптимізована конфігурація впровадила різноманітні типи індексів відповідно до цілей цієї роботи. Були створені наступні індекси.

Індекси B-дерев для пошуку рівності та діапазону:

```
CREATE INDEX idx_order_date ON orders(date);
CREATE UNIQUE INDEX idx_customer_email ON customers(email);
```

Складені індекси для перевірки чутливості до порядку стовпців:

```
CREATE INDEX idx_order_customer_date ON orders(customer_id, date);
CREATE INDEX idx_product_category_price ON products(category, price);
```

Індекси на основі функцій для підтримки обчислюваних предикатів:

```
CREATE INDEX idx_upper_name ON customers((upper(name)));
CREATE INDEX idx_order_month ON orders((date_trunc('month', date)));
```

Усі вимірювання були зібрані за допомогою EXPLAIN (ANALYZE, BUFFERS) з ідентичним обладнанням та налаштуваннями PostgreSQL. Щоб уникнути ефектів кешування, кожен тест починався з очищених буферів і повторювався тричі для кожної конфігурації, при цьому для аналізу

використовувалися середні значення. У табл. 1 подано порівняння між базовою та оптимізованою конфігураціями.

Таблиця 2.2. – Порівняння між базовою та оптимізованою конфігураціями

Query type	**Index type**	**Execution time, ms**		**Speedup (×)**	**Plan type**
		no index	**with index**		
Equality lookup (email)	B-tree	1425	3.2	**445**×	Seq Scan → Index Scan
Range query (date)	B-tree	5860	127	**46**×	Seq Scan → Index Range Scan
Join (customer_id)	Composite (customer_id, date)	9020	1320	**6.8**×	Hash Join → Nested Loop + Index Lookup
Aggregation (category)	Composite (category, price)	4870	2280	**2.1**×	Seq Scan → Bitmap Heap Scan
Sorting (date DESC)	B-tree	2740	18	**152**×	Sort → Index Scan (pre-ordered)
Case-insensitive search (upper(name))	Function-based	1460	24	**60**×	Seq Scan → Index Scan
Monthly aggregation (date_trunc('month', date))	Function-based	2150	105	**20**×	Seq Scan → Index Scan

Порівняльні результати продуктивності між базовою та оптимізованою конфігураціями підтвердили, що різні типи індексів забезпечують різні переваги залежно від вибірковості даних та складності запитів, таблиця 2.1.

Результати показують, що індекси B-дерев забезпечили найбільший приріст для запитів з високою вибірковістю, тоді як складені та функціональні індекси вирішували більш спеціалізовані випадки. Діапазонні запити, операції сортування та фільтри за рівністю отримали найбільший виграш завдяки ієрархічному механізму обходу B-дерев, який дозволив швидко звузити область

пошуку. Складені індекси забезпечили суттєве прискорення у запитах із багатостовпцевими предикатами, хоча їхня ефективність значною мірою залежала від порядку стовпців у визначенні індексу. Індекси на основі функцій істотно знизили вартість виконання запитів у тих випадках, коли обчислення над стовпцями в іншому разі перешкоджали б використанню звичайного індексу.

Наприклад, під час пошуку клієнтів за іменем без урахування регістру, базовий запит WHERE UPPER(name) = 'ALICE SMITH' примушував до послідовного сканування. Після створення індексу на (upper(name)) оптимізатор PostgreSQL зміг використовувати сканування індексу, скоротивши час виконання з 1,4 секунди до 24 мілісекунд. Аналогічно, час виконання щомісячної агрегації замовлень скоротився з 2,1 секунди до 0,1 секунди завдяки використанню індексу на виразі date_trunc('month', date), який забезпечив фільтрацію даних уже на рівні індексу.

На рис. 2.3 показано, як змінилося виконання запиту після індексації. На ньому показано перетворення для діапазонного запиту на Orders(date), де PostgreSQL замінив повне сканування таблиці прямим обходом структури B-дерева.

Рисунок 2.3 – Трансформація плану виконання після створення індексу

Ця трансформація відображає, як PostgreSQL реструктуризував виконання запитів після того, як індекси стали доступними. Замість зчитування всіх блоків таблиці, база даних виконувала вибіркове зчитування сторінок, зменшуючи як накладні витрати вводу-виводу, так і процесора. Перехід від послідовного до доступу на основі індексів скоротив загальний час виконання майже на два порядки.

Об’єднані експерименти демонструють, що ефективне проектування індексів не обмежується вибором між B-деревом або відсутністю індексу. Воно вимагає розуміння того, як розподіл даних, структура предикатів та використання функцій впливають на планування запитів. Індекси B-дерев забезпечували універсальні переваги для пошуку за рівністю та діапазоном. Складені індекси довели, що порядок стовпців важливий. Конфігурація (customer_id, date) підтримувала як фільтрацію, так і сортування, тоді як (date, customer_id) цього не робила, що призводило до менш ефективних планів. Індекси на основі функцій усунули неможливість саргажування виразів, перетворивши запити з обчисленнями на індексовані. У запитах з низькою вибірковістю, де багато рядків мають однакове значення, PostgreSQL використовував сканування купи бітових карт, ефективно поєднуючи кілька звернень до індексу в один ефективний шлях доступу.

Ці результати разом підтверджують теоретичну передумову, що вибір індексу має бути керований даними. Узгоджуючи тип та структуру індексу з поведінкою запитів, бази даних досягають значних покращень пропускної здатності та часу відгуку. Практичні експерименти довели, що навіть в межах однієї моделі індексування, варіація в кардинальності даних, порядку складених стовпців та доступі на основі функцій може суттєво змінити вибір оптимізатора та загальну продуктивність.

2.5. Результати та рекомендації

Результати експериментальної оцінки дозволяють інтерпретувати роль індексів не лише як допоміжних структур даних, але й як критичних компонентів, що фундаментально формують внутрішню поведінку оптимізатора запитів та визначають межі продуктивності реляційних систем баз даних. Порівнюючи плани виконання за базовими, B-деревовими, композитними та функціональними сценаріями, стає очевидним, що індекси служать основним механізмом, за допомогою якого PostgreSQL трансформує свою стратегію

виконання від вичерпного сканування до вибіркової навігації. Цей зсув змінює баланс між часом процесора, навантаженням на пам’ять та дисковим вводом/виводом, що призводить до суттєвого зменшення затримки виконання.

Покращення продуктивності, що спостерігаються в запитах на рівність та діапазон, зумовлені тим фактом, що структури B-дерев дозволяють виявляти ключі за логарифмічний час. Коли існує відповідний ключ, PostgreSQL проходить лише мінімальну кількість сторінок індексу, минаючи необхідність перевірки нерелевантних рядків. Така поведінка безпосередньо пояснює прискорення від кількох секунд до кількох мілісекунд, яке спостерігається в експериментах. Результати також підтверджують, що індекси B-дерев особливо ефективні, коли предикат запиту демонструє високу вибірковість, тобто лише невеликий відсоток рядків задовольняє умову. У цьому випадку різниця між послідовним скануванням та навігацією на основі індексів стає разючою, оскільки остання уникає торкання більшості блоків купи.

Аналіз складених індексів демонструє більш нюансовану картину продуктивності. Ці індекси за своєю суттю спрямовані, і крайній лівий стовпець визначає здатність індексу підтримувати операції пошуку, упорядкування та об'єднання. Коли крайній лівий стовпець був вирівняний з основною умовою фільтрації, оптимізатор міг повністю використовувати індекс, що призводило до суттєвого скорочення часу виконання. Однак, коли порядок індексів був зворотним, перевага в продуктивності значно зменшувалася, оскільки PostgreSQL не міг скористатися перевагами індексу для фільтрації та був змушений повернутися до послідовного сканування або планів растрових карт. Ці спостереження підкреслюють важливість проектування складених індексів з урахуванням реалістичних моделей робочого навантаження, оскільки теоретична оптимальність може відрізнятися від практичної ефективності залежно від того, як структуровані запити програми.

Експерименти, що включають індекси на основі функцій, додатково демонструють тісний зв’язок між структурою запиту та поведінкою оптимізатора. Коли стовпець вбудовано всередину функції в предикаті,

наприклад, у порівняннях без урахування регістру або групуванні на основі часу, PostgreSQL не може застосувати індекс, якщо індекс на основі виразів не відповідає функції. Результати вимірювань показують, що в базовому сценарії ці запити завжди запускали послідовне сканування, тоді як з індексами на основі функцій оптимізатор негайно застосовував сканування індексів. Така поведінка підтверджує, що індекси виразів – це не просто додаткові оптимізації, а важливі інструменти для забезпечення індексації запитів, що включають перетворення. Такі сценарії поширені в реальних системах, де нормалізація даних та гнучка фільтрація часто вимагають функцій або обчислень над стовпцями.

Поява повного сканування таблиці під час експериментів додає ще один аспект до розуміння поведінки оптимізатора. Ці стратегії виконання вказують на те, що PostgreSQL вибірково поєднує часткові збіги індексів, коли базовий розподіл даних або низька вибірковість роблять пряме сканування індексів неоптимальним. Цей гібридний підхід значно покращує продуктивність порівняно з повним послідовним скануванням, проте він не досягає такого ж прискорення, як високовибіркове сканування індексів. З практичної точки зору, поява повного сканування таблиці слід розглядати як сигнал про те, що запити можуть отримати користь від додаткового індексування або перегляду їхньої структури. Як альтернатива, вони можуть відображати притаманні обмеження розподілу даних, такі як наявність стовпців з низькою кардинальністю, де навіть оптимальне індексування не може повністю усунути накладні витрати вводу-виводу.

У сукупності результати всіх стратегій індексування демонструють чітке методологічне розуміння: оптимізація бази даних сильно залежить від взаємозв'язку між характеристиками даних та семантикою запитів. Одне рішення для індексування не може універсально задовольнити потреби в продуктивності всіх робочих навантажень. Натомість, до оптимізації слід підходити як до ітеративного, заснованого на доказах процесу, який включає аналіз шаблонів запитів, інтерпретацію планів виконання, проектування індексів, адаптованих до

цих шаблонів, та оцінку ефективності кожної стратегії індексування шляхом контрольованого тестування.

З практичної точки зору, отримані результати мають кілька важливих наслідків для адміністраторів баз даних та розробників. Перший висновок полягає в необхідності розуміння вибірковості запитів під час визначення того, які стовпці індексувати, оскільки високовибіркові стовпці дають найбільші переваги в продуктивності. Другий висновок стосується важливості узгодження складених індексних структур з фактичними предикатами запитів для забезпечення оптимального використання впорядкованих ключових сегментів. Третій висновок підкреслює цінність індексів на основі функцій для запитів, що включають обчислювані вирази, які в іншому випадку призводять до повного сканування, навіть коли базовий стовпець індексований. Нарешті, наявність сканування купи растрових зображень вказує на те, що певні робочі навантаження можуть вимагати переробки запитів або вибіркової денормалізації для досягнення подальших покращень, особливо коли фільтрація включає стовпці з обмеженою кардинальністю.

Загальні результати демонструють, що реляційні бази даних можуть досягти значного покращення продуктивності завдяки продуманому проекту індексів, причому приріст часто перевищує два порядки. Ці покращення не тільки прискорюють час відгуку програм, але й підвищують масштабованість базової системи, дозволяючи їй обробляти більші набори даних та вищий рівень паралельності без необхідності розширення обладнання. Аналіз підтверджує, що рішення щодо індексації повинні ґрунтуватися на емпіричних спостереженнях, а не на інтуїції, оскільки плани виконання та реальні вимірювання продуктивності є найнадійнішим показником ефективності будь-якої стратегії індексації.

Експерименти, проведені в цьому дослідженні, пропонують повне розуміння того, як стратегії індексації впливають на внутрішню роботу PostgreSQL та як ці стратегії можна систематично застосовувати для оптимізації продуктивності бази даних. Висновки, отримані в результаті аналізу, служать практичним посібником для проектування індексів та забезпечують міцну основу для майбутніх зусиль з оптимізації.

ВИСНОВКИ

Дослідження, представлене в кваліфікаційній роботі, пропонує комплексне вивчення ролі індексації як фундаментального механізму продуктивності в реляційних системах управління базами даних. Інтегруючи теоретичний аналіз з методологічно структурованою експериментальною оцінкою, дослідження продемонструвало, як стратегії індексації безпосередньо впливають на ефективність, масштабованість та швидкість реагування програм на основі SQL. Протягом усієї роботи PostgreSQL слугував репрезентативною сучасною СУБД, що дозволило дослідникам дослідити поведінку індексації в реальному середовищі виконання, зберігаючи при цьому узагальнюваність для інших реляційних систем.

Теоретичний компонент встановив фундаментальне розуміння того, що індекси, особливо ті, що базуються на структурах B-дерев, розроблені для мінімізації обсягу даних, які система повинна прочитати для відповіді на запит. Їхній ієрархічний механізм пошуку ефективно зменшує кількість звернень до сторінок і дозволяє швидко знаходити значення навіть у наборах даних, що містять мільйони записів. Теоретичний аналіз також пояснив, чому складені індекси залежать від упорядкування ключових атрибутів, як індекси на основі функцій підтримують запити, що містять обчислювані вирази, і чому певні операції за своєю суттю опираються оптимізації, якщо не введено спеціалізовані структури індексації. Ці концептуальні висновки створили основу для прогнозування того, як різні стратегії індексації поводяться під час експериментального тестування.

Емпірична складова кваліфікаційної роботи не лише підтвердила теоретичні очікування, але й розширила їх, виявивши поведінку, яку можна спостерігати лише в реальному середовищі. Побудова великомасштабного набору даних, що містить п'ятдесят тисяч клієнтів, десять тисяч товарів, сімсот тисяч замовлень та два мільйони деталей замовлення, забезпечила реалістичний контекст для вивчення продуктивності запитів. Базові вимірювання без

вторинних індексів виявили неефективність послідовного сканування: навіть запити, що фільтруються за одним унікальним атрибутом, були змушені проходити через усю таблицю. Ці результати проілюстрували, як у великих базах даних відсутність індексації стає домінуючим вузьким місцем продуктивності, що робить складні робочі навантаження непрактичними.

Після введення індексів поведінка системи різко змінилася. Індекси B-дерев за атрибутами з високою вибірковістю призвели до покращення продуктивності в кілька сотень разів, скоротивши час виконання з секунд до мілісекунд. Діапазонні запити за датами продемонстрували важливість упорядкованого проходження індексів, коли механізм бази даних сканував лише відповідну частину індексу, а не весь набір даних. Складені індекси довели, що індексація повинна точно відповідати структурі реальних запитів; коли порядок стовпців відображав фактичні моделі використання, продуктивність суттєво покращувалася. І навпаки, коли порядок стовпців не відповідав предикатам, які зазвичай використовуються програмою, переваги ставали незначними. Індекси на основі функцій ілюстрували не менш важливий принцип, дозволяючи фільтрацію на рівні індексу, навіть коли запити застосовували перетворення до стовпців. Без таких індексів база даних постійно поверталася до повного сканування, що підтверджує, що індексація виразів є важливою для підтримки гнучких шаблонів пошуку та агрегації.

Поява повного сканування таблиці у сценаріях із низькою вибірковістю надала додаткове розуміння поведінки запитів. Хоча PostgreSQL не підтримує виділені структури індексів бітових карт, його оптимізатор може комбінувати кілька умов індексу, використовуючи логіку бітових карт. Ці плани виконання діяли як проміжна стратегія оптимізації, особливо ефективна при фільтрації за атрибутами з невеликою кількістю різних значень. Їх поява демонструє, що оптимізатор динамічно вибирає між скануванням індексів, скануванням бітових карт та послідовним скануванням, використовуючи моделі на основі вартості, які відображають як складність запитів, так і розподіл даних. Ця нюансована

поведінка підкреслює важливість розуміння не лише створення індексів, але й того, як оптимізатор інтерпретує та вибирає стратегії виконання.

Ширша інтерпретація експериментальних результатів свідчить про те, що проектування ефективних індексів вимагає більше, ніж технічних знань команд SQL. Це вимагає систематичного та аналітичного підходу, що ґрунтується на розумінні розподілу даних, аналізі реальних робочих навантажень та інтерпретації планів виконання. Дослідження підкреслює, що жодна єдина конфігурація індексу не може ефективно обслуговувати всі робочі навантаження. Натомість, індексування має бути адаптоване до реалістичних шаблонів запитів, балансуючи покращення продуктивності читання з потенційними накладними витратами на запис. Дослідження демонструє, що індексування – це не просто технічне вдосконалення, а стратегічне рішення, яке впливає на надійність, пропускну здатність та довгострокову масштабованість системи баз даних.

З практичної точки зору, результати пропонують змістовні рекомендації для розробників та адміністраторів баз даних. Запити, які значною мірою залежать від пошуку рівності, досягають найбільшого приросту продуктивності від одностовпцевих B-деревоподібних структур.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. De Oliveira, V.F.; Pessoa, M.A.d.O.; Junqueira, F.; Miyagi, P.E. SQL and NoSQL Databases in the Context of Industry 4.0. Machines 2022, 10, 20. https://doi.org/10.3390/machines10010020
2. Bader, S.; Barnstedt, E.; Bedenbender, H.; Billman, M.; Boss, B.; Braunmandl, A. Details of the Asset Administration Shell Part 1—The Exchange of Information between Partners in the Value Chain of Industrie 4.0; Federal Ministry for Economic Affairs and Energy (BMWi): Berlin, Germany, 2019.
3. Ziheng Wei and Sebastian Link. 2021. Embedded Functional Dependencies and Data-completeness Tailored Database Design. ACM Trans. Database Syst. 46, 2, Article 7 (June 2021), 46 pages. https://doi.org/10.1145/3450518
4. Baoqing Cai, Yu Liu, Lin Ma, Pingqi Huang, Bingcheng Lian, Ke Zhou, Jia Yuan, Jie Yang, Xiaofan Cai, and Peijun Wu. 2025. SCompression: Enhancing Database Knob Tuning Efficiency Through Slice-Based OLTP Workload Compression. Proc. VLDB Endow. 18, 6 (February 2025), 1865–1878. https://doi.org/10.14778/3725688.3725712
5. N. El-Sayed, Z. Sun, K. Sun and R. Mayerhofer, "OLTP In Real Life: A Large-scale Study of Database Behavior in Modern Online Retail," 2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Houston, TX, USA, 2021, pp. 1-8, doi: 10.1109/MASCOTS53633.2021.9614295.
6. Karri N, Pedda Muntala PSR. Query Optimization Using Machine Learning. IJETCSIT [Internet]. 2023 Dec. 30 [cited 2025 Nov. 14];4(4):109-17. Available from: https://www.ijetcsit.org/index.php/ijetcsit/article/view/411
7. G. Theodorakis, J. Clarkson and J. Webber, "An Empirical Evaluation of Variable-length Record B+Trees on a Modern Graph Database System," 2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW), Utrecht, Netherlands, 2024, pp. 343-349, doi: 10.1109/ICDEW61823.2024.00051.

8. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, “Cypher: An evolving query language for property graphs,” in SIGMOD, 2018.
9. Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3448016.3457245
10. Panagiotis Antonopoulos, Mansi Chauhan, Shailender Dabas, Rajat Jain, Darshan Kattera, Wonseok Kim, Hanuma Kodavalla, Nikolas Ogg, Prashanth Purnananda, Rahul Ranjan, Alex Swanson, and Divyesh Tikmani. 2025. MD-MVCC: Multi-Version Concurrency Control for Schema Changes in Azure SQL Database. Proc. VLDB Endow. 18, 12 (August 2025), 4791–4803. https://doi.org/10.14778/3750601.3750605
11. Neha and Banita, "High-Performance Concurrency Control in Multi-User Database Systems: Analysis and Optimization Strategies," 2024 3rd Edition of IEEE Delhi Section Flagship Conference (DELCON), New Delhi, India, 2024, pp. 1-5, doi: 10.1109/DELCON64804.2024.10866991.
12. Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-optimized multi-version concurrency control for disk-based database systems. Proc. VLDB Endow. 15, 11 (July 2022), 2797–2810. https://doi.org/10.14778/3551793.3551832
13. Luca Gretscher and Jens Dittrich. 2025. How to Optimize SQL Queries? A Comparison Between Split, Holistic, and Hybrid Approaches. Proc. VLDB Endow. 18, 11 (July 2025), 3910–3922. https://doi.org/10.14778/3749646.3749663
14. Abbasi, M.; Bernardo, M.V.; Váz, P.; Silva, J.; Martins, P. Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. Information 2024, 15, 429. https://doi.org/10.3390/info15080429

15. Shahrokhi, H.; Shaikhha, A. An Efficient Vectorized Hash Table for Batch Computations. In 37th European Conference on Object-Oriented Programming (ECOOP 2023); Schloss-Dagstuhl-Leibniz Zentrum für Informatik: Wadern, Germany, 2023; pp. 27:1–27:27.
16. Goetz Graefe (2024), "More Modern B-Tree Techniques", Foundations and Trends® in Databases: Vol. 13: No. 3, pp 169-249. http://dx.doi.org/10.1561/1900000070
17. Ziaya, I., Farou, B., Kouahla, Z. et al. A blockchain and B-tree indexing approach to improve traceability and query efficiency in food supply chain. Cluster Comput 28, 1027 (2025). https://doi.org/10.1007/s10586-025-05631-3
18. Kaisong Huang, Yuliang He, and Tianzheng Wang. 2022. The past, present and future of indexing on persistent memory. Proc. VLDB Endow. 15, 12 (August 2022), 3774–3777. https://doi.org/10.14778/3554821.3554897
19. Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing. Proc. VLDB Endow. 18, 2 (October 2024), 399–412. https://doi.org/10.14778/3705829.3705854
20. Ashwini Mandale, Neeraj Sharma, Design and Development of Schema for Schemaless Databases, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 10.1142/S0218488525500011, 33, 01, (1-28), (2025).
21. Tabrizchi, H. (2025). Optimizing Your Script with Indexes and Views. In: Narrative SQL. Apress, Berkeley, CA. https://doi.org/10.1007/979-8-8688-1560-7_9
22. Kuhn, D., Kyte, T. (2022). Database Tables. In: Expert Oracle Database Architecture. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-7499-6_10
23. Zhao, R.; Chen, G.; Zhang, Z.; Luo, W. Progressive Collapse Resistance Assessment of a Multi-Column Frame Tube Structure with an Assembled Truss Beam Composite Floor under Different Column Removal Conditions. Buildings 2024, 14, 111. https://doi.org/10.3390/buildings14010111

24. Martins, P., Tomé, P., Wanzeller, C., Sá, F., Abbasi, M. (2021). Comparing Oracle and PostgreSQL, Performance and Optimization. In: Rocha, Á., Adeli, H., Dzemyda, G., Moreira, F., Ramalho Correia, A.M. (eds) Trends and Applications in Information Systems and Technologies . WorldCIST 2021. Advances in Intelligent Systems and Computing, vol 1366. Springer, Cham. https://doi.org/10.1007/978-3-030-72651-5_46
25. X. Chen et al., "150 m/500 Mbps Underwater Wireless Optical Communication Enabled by Sensitive Detection and the Combination of Receiver-Side Partial Response Shaping and TCM Technology," in Journal of Lightwave Technology, vol. 39, no. 14, pp. 4614-4621, July15, 2021, doi: 10.1109/JLT.2021.3077086.
26. Y. Deng, "Recommender Systems Based on Graph Embedding Techniques: A Review," in IEEE Access, vol. 10, pp. 51587-51633, 2022, doi: 10.1109/ACCESS.2022.3174197.
27. Maxime Schoemans, Walid G. Aref, Esteban Zimányi, and Mahmoud Sakr. 2024. Multi-Entry Generalized Search Trees for Indexing Trajectories. In Proceedings of the 32nd ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL '24). Association for Computing Machinery, New York, NY, USA, 421–431. https://doi.org/10.1145/3678717.3691320
28. Wekesa, E.N.; DeCusatis, C.; Zhu, A. A Black Box Comparison of Machine Learning Reverse Image Search for Cybersecurity OSINT Applications. Electronics 2023, 12, 4822. https://doi.org/10.3390/electronics12234822
29. Douglas B. Rumbaugh, Dong Xie, and Zhuoyue Zhao. 2024. Towards Systematic Index Dynamization. Proc. VLDB Endow. 17, 11 (July 2024), 2867–2879. https://doi.org/10.14778/3681954.3681969
30. Jiong Xie, Zhen Chen, Jianwei Liu, Fang Wang, Feifei Li, Zhida Chen, Yinpei Liu, Songlu Cai, Zhenhua Fan, Fei Xiao, and Yue Chen. 2022. Ganos: a multidimensional, dynamic, and scene-oriented cloud-native spatial database engine. Proc. VLDB Endow. 15, 12 (August 2022), 3483–3495. https://doi.org/10.14778/3554821.3554838

31. J. Bariffi, H. Bartz, G. Liva and J. Rosenthal, "Error-Correction Performance of Regular Ring-Linear LDPC Codes Over Lee Channels," in IEEE Transactions on Information Theory, vol. 70, no. 11, pp. 7820-7839, Nov. 2024, doi: 10.1109/TIT.2024.3436938.
32. F. Gao, W. Chen, M. Wu and J. She, "Output-Feedback Control in Nonlinear Singular Systems With Exogenous Disturbance Based on State Decomposition Technique," in IEEE Transactions on Industrial Informatics, vol. 20, no. 2, pp. 1952-1962, Feb. 2024, doi: 10.1109/TII.2023.3283543.
33. Francisquez M, Juno J, Hakim A, Hammett GW, Ernst DR. Improved multispecies Dougherty collisions. Journal of Plasma Physics. 2022;88(3):905880303. doi:10.1017/S0022377822000289
34. Maroua Masmoudi, Sana Ben Abdallah Ben Lamine, Mohamed Hedi Karray, Bernard Archimede, and Hajer Baazaoui Zghal. 2024. Semantic Data Integration and Querying: A Survey and Challenges. ACM Comput. Surv. 56, 8, Article 209 (August 2024), 35 pages. https://doi.org/10.1145/3653317
35. X. Cai, W. Xu, S. Hong, L. Wang and L. Zhang, "General Carrier Index Aided Dual-Mode Differential Chaos Shift Keying With Full Mapping: Design and Optimization," in IEEE Transactions on Vehicular Technology, vol. 70, no. 11, pp. 11665-11677, Nov. 2021, doi: 10.1109/TVT.2021.3113315.

ДОДАТКИ

ДОДАТОК А

Генерація даних

```
import csv
import random
from datetime import datetime, timedelta
from faker import Faker
import math

SEED = 20251113
random.seed(SEED)
fake = Faker()
Faker.seed(SEED)

# ----- volumes -----
N_CUSTOMERS = 50_000
N_PRODUCTS = 10_000
N_ORDERS = 700_000
N_ORDER_DETAILS = 2_000_000

# ----- categories and weights -----
categories = [
    "Books", "Electronics", "Home", "Toys", "Grocery", "Health",
    "Beauty", "Fashion", "Sports", "Automotive", "Garden", "Office"
]
category_weights = [18,14,10,8,8,7,7,7,7,7,4,3]

# ----- helper: lognormal prices -----
def generate_price():
    # log-normal with mu=3.6, sigma=0.75, capped at 9999.99
    val = math.exp(random.gauss(3.6, 0.75))
    return round(min(val, 9999.99), 2)
```

```
    # ----- generate customers -----
    with open("customers.csv", "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id", "name", "city", "email"])
        for i in range(1, N_CUSTOMERS + 1):
            name = fake.name()
            city = fake.city()
            email = f"user{i}@example.com"
            writer.writerow([i, name, city, email])

    # ----- generate products -----
    with open("products.csv", "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id", "name", "category", "price"])
        for i in range(1, N_PRODUCTS + 1):
            category            =            random.choices(categories,
weights=category_weights, k=1)[0]
            name = f"{category} item {i}"
            price = generate_price()
            writer.writerow([i, name, category, price])


    def generate_date():
        start_date = datetime(2022, 1, 1)
        end_date = datetime(2025, 10, 31, 23, 59, 59)
        total_days = (end_date - start_date).days
        day_offset = random.randint(0, total_days)
        date = start_date + timedelta(days=day_offset)
        # add weekly & yearly seasonal weight
        week_day = date.weekday()
        seasonal = 1.0
        if week_day >= 4:  # Fri–Sun
            seasonal *= 1.25
```

```
            if date.month in (11, 12):
                seasonal *= 1.35
            # apply small random shift within the day
            seconds = random.randint(0, 86400)
            date = date + timedelta(seconds=seconds)
            return date

        # ----- generate orders -----
        with open("orders.csv", "w", newline="", encoding="utf-8") as f:
            writer = csv.writer(f)
            writer.writerow(["id", "customer_id", "date", "amount"])
            for i in range(1, N_ORDERS + 1):
                cust_id = random.randint(1, N_CUSTOMERS)
                dt = generate_date().strftime("%Y-%m-%d %H:%M:%S")
                writer.writerow([i, cust_id, dt, 0.00])  # amount will be
updated later

        with open("order_details.csv", "w", newline="", encoding="utf-8")
as f:
            writer = csv.writer(f)
            writer.writerow(["order_id", "product_id", "quantity"])
            order_ids = list(range(1, N_ORDERS + 1))
            product_ids = list(range(1, N_PRODUCTS + 1))

            total_rows = 0
            while total_rows < N_ORDER_DETAILS:
                order_id = random.choice(order_ids)
                n_lines = random.randint(2, 7)  # 2–7 lines per order
                for _ in range(n_lines):
                    if total_rows >= N_ORDER_DETAILS:
                        break
                    product_id = random.choice(product_ids)
                    quantity          =          random.choices([1,2,3,4,5],
weights=[55,25,12,6,2])[0]
```

```
            writer.writerow([order_id, product_id, quantity])
            total_rows += 1

print("Data generation completed successfully:")
print(f"  customers.csv  → {N_CUSTOMERS} rows")
print(f"  products.csv   → {N_PRODUCTS} rows")
print(f"  orders.csv     → {N_ORDERS} rows")
print(f"  order_details.csv → {N_ORDER_DETAILS} rows")
```

ДОДАТОК В

Створення індексу

```
-- B-tree indexes
CREATE UNIQUE INDEX idx_customer_email
    ON customers(email);

CREATE INDEX idx_order_date
    ON orders(date);

CREATE INDEX idx_order_date_desc
    ON orders(date DESC);

-- Composite indexes
CREATE INDEX idx_order_customer_date
    ON orders(customer_id, date);

CREATE INDEX idx_product_category_price
    ON products(category, price);

-- Function-based indexes
CREATE INDEX idx_upper_name
    ON customers((UPPER(name)));

CREATE INDEX idx_order_month
    ON orders((date_trunc('month', date)));

-- Partial index example
CREATE INDEX idx_products_active
    ON products(price)
    WHERE price > 0;
```

ABSTRACT

Wang Chao. Optimization of queries in relational databases using indices. - Manuscript. Manuscript. Qualification work for the degree of "Master" in the field of Computer Science, educational program "Computer Science and Information Technologies." – Lesya Ukrainka Volyn National University. – 2025

Introduction. The rapid growth of data volumes in modern information systems has intensified the demand for efficient query processing in relational databases. As organizations accumulate millions of transactional records, traditional sequential scans become increasingly impractical, causing delays in analytical workflows and limiting system responsiveness. Indexing remains one of the most reliable optimization mechanisms available in relational database management systems, yet its effectiveness depends strongly on correct design, selectivity, data distribution, and workload characteristics. The research investigates how indexing strategies influence query performance, with particular attention to B-tree, composite, and function-based indexes in PostgreSQL.

The purpose of this research is to examine the impact of different index types on the performance of relational database queries and to identify practical indexing strategies that provide measurable improvements for real-world workloads.

Results. Baseline measurements showed that queries without indexes consistently triggered sequential scans, resulting in high latency and significant I/O load. After applying B-tree, composite, and expression-based indexes, execution plans transitioned to index scans and index-only scans, reducing query time from seconds to milliseconds. Range queries benefited from ordered B-tree traversal, composite indexes demonstrated sensitivity to column order, and function-based indexes enabled optimization of predicates containing transformations.

Conclusions. The findings confirm that indexing is a decisive factor in improving performance of relational database systems. Properly designed indexes significantly accelerate data retrieval, reduce system load, and increase overall scalability without requiring hardware upgrades.

Keywords: relational databases, indexing, B-tree index, composite index, function-based index, PostgreSQL, query optimization.

АНОТАЦІЯ

Ван Чао. Оптимізація запитів у реляційних базах даних із використанням індексів – Рукопис. Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки, освітньої програми Комп'ютерні науки та інформаційні технології. – Волинський національний університет імені Лесі Українки. – 2025 р.

Вступ. Швидке зростання обсягів даних у сучасних інформаційних системах значно підвищило потребу в ефективній обробці запитів у реляційних базах даних. Оскільки організації накопичують мільйони транзакційних записів, традиційні послідовні сканування стають дедалі менш практичними, що призводить до затримок у аналітичних процесах і обмежує швидкодію системи. Індексування залишається одним із найнадійніших механізмів оптимізації в системах управління реляційними базами даних, проте його ефективність значною мірою залежить від правильного проєктування, вибірковості, розподілу даних та характеристик навантаження. У роботі вивчається вплив стратегій індексування на продуктивність запитів, зокрема увага приділяється B-деревам, складеним індексам і індексам на основі функцій у PostgreSQL.

Мета дослідження полягає у вивченні впливу різних типів індексів на продуктивність запитів у реляційних базах даних та визначенні практичних стратегій індексування, які забезпечують відчутне покращення продуктивності у реальних сценаріях навантаження.

Результати. Базові вимірювання показали, що запити без індексів систематично викликали послідовні сканування, що призводило до високої затримки та значного навантаження на систему. Після застосування B-дерев, складених та виразних індексів плани виконання перейшли на сканування по індексу та виключно по індексу (index-only scans), що дозволило скоротити час виконання запитів із секунд до мілісекунд. Діапазонні запити отримали вигоду від впорядкованого обходу B-дерева, складені індекси продемонстрували чутливість до порядку стовпців, а індекси на основі функцій дозволили оптимізувати предикати з трансформаціями.

Висновки. Результати підтверджують, що індексування є визначальним фактором у підвищенні продуктивності реляційних баз даних. Правильно спроєктовані індекси значно прискорюють отримання даних, зменшують навантаження на систему та підвищують загальну масштабованість без потреби у модернізації обладнання.

Ключові слова: реляційні бази даних, індексування, B-дерево, складений індекс, індекс на основі функцій, PostgreSQL, оптимізація запитів.