Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Порівняння принципів роботи індексів у PostgreSQL та MySQL

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи
Яремко С.А.

_____
(підпис)

«____» _____ 2023 р.

Виконала студентка
Золотар А.С.

«____» _____ 2023 р.

Київ 2023

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Завідувач кафедри інформатики
Гороховський С.С.
«____» _____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентці Золотар Анастасії Сергіївні факультету інформатики 3-го курсу

ТЕМА: Порівняння принципів роботи індексів у PostgreSQL та MySQL

Зміст ТЧ до курсової роботи:
Вступ
1. Теоретичні відомості
2. Реалізація індексів у PostgreSQL
3. Реалізація індексів у MySQL
4. Відмінності індексів у PostgreSQL і MySQL
5. Проектна реалізація
Висновки

Дата видачі «____» _____ 2023 р.

Керівник _____

(*підпис)*

Завдання отримав _____

(*підпис)*

Календарний план виконання роботи

№ п/п	Назва етапу курсової роботи	Термін виконання	Примітки
1.	Отримання теми курсової роботи	18.10.2022	
2.	Огляд літератури за темою роботи	листопад 2022 р.	
3.	Ознайомлення з теоретичним матеріалом	грудень 2022 р.	
4.	Написання першого розділу курсової роботи	15.01.2023	
5.	Написання другого розділу курсової роботи	12.02.2023	
6.	Написання третього розділу курсової роботи	20.03.2023	
7.	Проєктування та створення демонстраційних баз даних	04.04.2023	
8.	Написання четвертого розділу курсової роботи	10.04.2023	
9.	Проєктування та створення застосунку	23.04.2023	
10.	Написання п’ятого розділу курсової роботи	10.05.2023	
11.	Завершення написання та оформлення курсової роботи	11.05.2023	
12.	Подання роботи для перевірки на плагіат	18.05.2023	
13.	Захист курсової роботи	24.05.2023	

Виконавець курсової роботи Золотар Анастасія Сергіївна

Науковий керівник Яремко Соломія Андріївна

«_____»_____

Зміст

Зміст 4
Анотація 5
Вступ 6
Розділ 1. Теоретичні відомості 8
1.1. Основні поняття 8
1.2. Індекс на основі B+-дерева 11
1.3. Індекс на основі хеш-таблиці 13
Розділ 2. Реалізація індексів у PostgreSQL 15
2.1. Короткий огляд PostgreSQL 15
2.2. Типи індексів у PostgreSQL 15
2.3. Застосування індексів у PostgreSQL 19
Розділ 3. Реалізація індексів у MySQL 21
3.1. Короткий огляд MySQL 21
3.2. Типи індексів у MySQL 21
3.3. Застосування індексів у MySQL 23
Розділ 4. Відмінності індексів у PostgreSQL і MySQL 25
4.1. Порівняння ефективності читання та оновлення даних 25
4.2. Зіставлення основних можливостей та розширень 25
Розділ 5. Проектна реалізація 28
5.1. Проектування бази даних 28
5.2. Проектування застосунку 29
5.3. Аналіз виконання запитів 29
Висновки 35
Список джерел та літератури 36

Анотація

Курсова робота присвячена дослідженню та порівнянню особливостей реалізації та застосування індексів у PostgreSQL і MySQL. Розглянуто основні теоретичні аспекти, пов’язані з індексами. Наведено опис наявних у двох системах типів індексів, їхніх переваг та обмежень. На прикладах проаналізовано та зіставлено підходи оптимізаторів PostgreSQL і MySQL у пришвидшенні виконання запитів.

Вступ

Попри стрімкий розвиток ресурсів та можливостей апаратного забезпечення, важливість ефективного виконання програм залишається актуальною, адже обсяг інформації, який потрібно опрацьовувати, зростає ще швидше. База даних – це ключовий елемент будь-якої інформаційної системи, тому, розглядаючи способи пришвидшення роботи програми, насамперед варто подбати про ефективне виконання операцій з даними.

Одним із найбільш потужних засобів оптимізації є індекси – структури, що дають змогу швидко здійснювати пошук потрібних значень під час виконання запитів до бази даних. Утім, як будь-які допоміжні інструменти, індекси мають недоліки, що передусім виявляються за їхнього некоректного використання, зумовленого браком розуміння як загальних структур та алгоритмів, які використовуються для створення та підтримання індексів, так і особливостей їхньої реалізації у конкретних СКБД.

Дослідження принципів імплементації індексів дає змогу коректно застосовувати їх в ефективний спосіб, зменшуючи вплив додаткових витрат на їхнє підтримання. Водночас, вивчення та порівняння відмінностей у механізмах роботи індексів у різних СКБД допомагає робити ґрунтовніший вибір під час проєктування системи, а також уникнути потенційних проблем, викликаних особливостями або обмеженнями їхньої конкретної реалізації.

PostgreSQL і MySQL – дві з найбільш поширених на сьогодні реляційних систем керування базами даних, які протягом десятків років існування продемонстрували надійність і швидкість своєї роботи. Водночас, вони використовують різні підходи в оптимізації запитів, які мають як переваги, так і недоліки.

Метою курсової роботи є дослідження та порівняння принципів, закладених в основу функціонування індексів у PostgreSQL і MySQL, зокрема, розгляд технічних особливостей двох реалізацій і зіставлення ефективності їхнього застосування. Завдання роботи також полягає в описі наявних різновидів індексів та аналізі їхніх переваг, недоліків та обмежень в оптимізації запитів на прикладах.

Текстова частина курсової роботи складається з п’ятьох розділів.

У першому розділі представлено основні теоретичні поняття, пов’язані з індексами. Зокрема, наведено короткий опис способу збереження даних на дисках, а також причини, які зумовлюють потребу в оптимізації виконання запитів; розглянуто найпоширеніші структури індексів – B+-дерева та хеш-індекси.

Другий та третій розділи присвячено детальному дослідженню особливостей реалізації індексів у PostgreSQL і MySQL відповідно: описано наявні типи індексів, а також можливості та обмеження їхнього застосування.

У четвертому розділі зіставлено основні відмінності у реалізації та принципах роботи індексів у PostgreSQL і MySQL, наведено порівняльні таблиці.

У п’ятому розділі наведено опис практичної реалізації застосунку, який використовує базу даних з великим обсягом записів. На прикладі декількох запитів, які використовуються у застосунку, проаналізовано та порівняно підходи оптимізаторів PostgreSQL і MySQL у застосуванні індексів для пришвидшення пошуку даних.

Розділ 1. Теоретичні відомості

1.1. Основні поняття

Більшість СКБД зберігає персистентні дані на диску і за потреби опрацювання зчитує їх до оперативної пам’яті. Таблицям бази даних відповідають окремі файли операційної системи, що логічно поділені на сторінки фіксованого розміру, які, зі свого боку, можуть містити декілька записів даних. Дискові операції введення та виведення працюють не з окремими записам, а зі сторінками.

Вартість виконання дискових операцій є високою, тому система виконує оптимізацію для зменшення їхньої кількості. Це завдання покладено на компонент СКБД, який називається оптимізатор. Деталі роботи оптимізатора значно залежать від конкретної системи, але в будь-якому випадку його основним завданням є отримання доступу до даних у найбільш ефективний спосіб, використовуючи зібрану для даних статистику. Зокрема, оптимізатор приймає рішення, чи доцільним буде використання індексу, якщо такий був створений для таблиці, до якої здійснюється звернення у запиті [1, с. 30].

Індекси – це структури даних, які дають змогу ефективно виконувати операцію отримання значень за певним ключем, не вдаючись до сканування всієї таблиці. Стовпець або множину стовпців, за значеннями яких здійснюють пошук записів у файлі, називають ключем пошуку. Кожен індекс побудований на основі такого ключа. Як приклад, розглянемо запит:

```
SELECT first_name, last_name
FROM clients
WHERE email = ‘student@ukma.edu.ua’;
```

Якщо для таблиці clients не було створено індексу за ключем пошуку email, то в загальному випадку системі доведеться виконувати операцію зчитування всіх сторінок файлу, у якому розміщено дані clients, та проглядати кожен запис у них,

щоб знайти саме той, який має вказане в запиті значення email. Застосування ж індексу в цій ситуації дало б змогу уникнути сканування всієї таблиці та пришвидшити виконання запиту.

Індекси зазвичай представлені окремими файлами, кожен запис яких містить значення ключа пошуку й указник на відповідний рядок файлу з даними. Найчастіше указник ідентифікує сторінку, на якій розміщено відповідні дані, і відступ до нього у її межах [2, с. 626].

Існує два загальні різновиди індексів: упорядковані та хеш-індекси. Записи у впорядкованому індексі зберігаються відсортованими за значенням ключа пошуку. Індекс, порядок записів якого збігається з порядком записів у файлі з даними, називають кластерним або первинним. Очевидно, що будь-яка таблиця може мати лише один кластерний індекс. Відповідно, решту індексів, у яких порядок ключів відмінний від упорядкування записів із даними, називають некластерними або вторинними. Отож один запис із даними можна ідентифікувати за допомогою різних індексів.

Упорядкований індекс може бути щільним або розрідженим. Щільним називають той індекс, який містить запис для кожного рядка з даними таблиці. З іншого боку, розріджений індекс містить записи лише для частини рядків. Найчастіше розріджений індекс має по одному указнику для кожної сторінки з даними [3, с. 622]. Це дає змогу зменшити розмір файлу, але збільшує час пошуку потрібного запису.

Також індекси буваю багаторівневими. Для таблиць зі значним обсягом даних сам індекс теж може містити надто велику кількість записів, щоб швидко здійснювати пошук серед них. У цьому випадку можна створити індекс другого рівня, тобто “індекс для індексу”. А якщо і він досить великий, тоді ще й індекс третього рівня і т.д.

Деякі СКБД розміщують записи з даними безпосередньо у структурі індексу, не створюючи окремих файлів таблиць. Такий індекс є кластерним за визначенням. За такого підходу записи вторинних індексів можуть містити не указники на розташування відповідних рядків у файлі, а значення ключів пошуку кластерного індексу, найчастіше якими є первинні ключі. Це дає змогу зменшити кількість операцій модифікації вторинних індексів у випадку оновлення даних, але, водночас, це призводить до збільшення вартості читання.

Індекси забезпечують чи не найпотужніший спосіб підвищення ефективності виконання запитів. Хоча деталі роботи індексів значно залежать від конкретної СКБД, всі вони використовують подібні принципи та структури.

Утім, варто розуміти, що, пришвидшуючи читання записів, індекси також вимагають додаткових витрат на своє підтримання. Під час виконання операцій вставки, оновлення чи видалення даних таблиці її індекси, якщо такі були створені, повинні бути перебудовані, причому в межах однієї транзакції.

Бази даних невеликого обсягу можуть ефективно працювати і без використання індексів, але в міру збільшення записів у таблиці час виконання запитів помітно довшає. Некоректне застосування індексів неодмінно призводить до результату, протилежного їхній головній меті, а саме до погіршення ефективності опрацювання даних. Тому важливо розуміти структури та принципи, закладені в основу реалізації індексів.

Для побудови індексів сучасні СКБД можуть використовувати різноманітні структури даних. Найпоширеніше застосування мають індекси на основі B+-дерев і хеш-таблиць.

1.2. Індекс на основі B+-дерева

B+-дерево є узагальненням збалансованого дерева бінарного пошуку, у якому вузли можуть мати вищий степінь розгалуження і, відповідно, меншу висоту. B+-дерево складається з трьох рівнів: кореня, внутрішніх вузлів та листкових вузлів. Оскільки B+-дерева використовуються для організації та навігації сторінками фіксованого розміру, терміни вузол та сторінка в цьому контексті часто є взаємозамінними [4, с. 35].

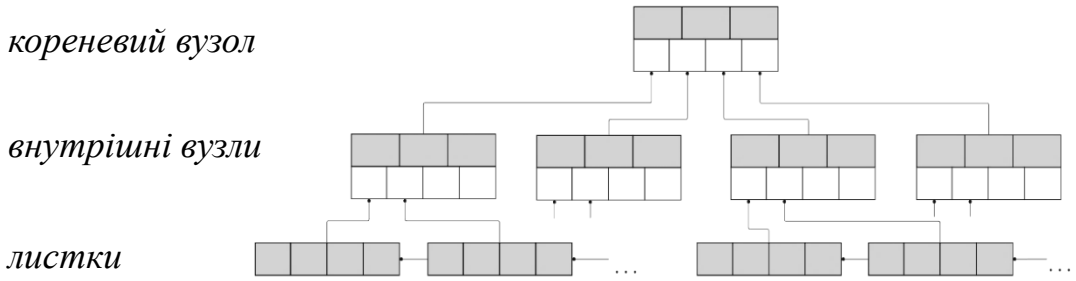

Рисунок 1.1. Структура B+-дерева

Розглянемо структуру B+-дерева як її визначають Г. Гарсіа-Моліна, Дж. Ульман і Дж. Уідом [3, с. 634-635].

- Будь-яке B+-дерево характеризується параметром n. Кожен вузол B+-дерева може вміщувати n значень ключів та $[n + 1]$ указників.
- Кореневий вузол завжди містить щонайменше $[n \,/\, 2]$ указники, якщо саме дерево не складається лише з одного вузла.
- Щонайменше $[(n + 1) \,/\, 2]$ з $[n + 1]$ указників у внутрішніх вузлах вказують на вузли нижчого рівня. Указник P_1 на Рисунку 1.2 адресує те піддерево, значення всіх ключів K_s якого менше за K_1. Указник P_2 посилається на піддерево з ключами, не меншими за K_1 і строго більшими за K_2 і т.д. Нарешті, указник P_n адресує піддерево, ключі якого дорівнюють або більші за K_{n-1}.

- Ключі у листках розміщені в порядку зростання. Останній указник у листку завжди вказує на наступний по порядку вузол. Щонайменше $[(n + 1) / 2]$ з n указників у листку використані для посилання на записи даних; решта вважаються вільними.

У такий спосіб нелисткові вузли B+-дерева формують багаторівневий розріджений індекс для листкових вузлів [2, с. 635].

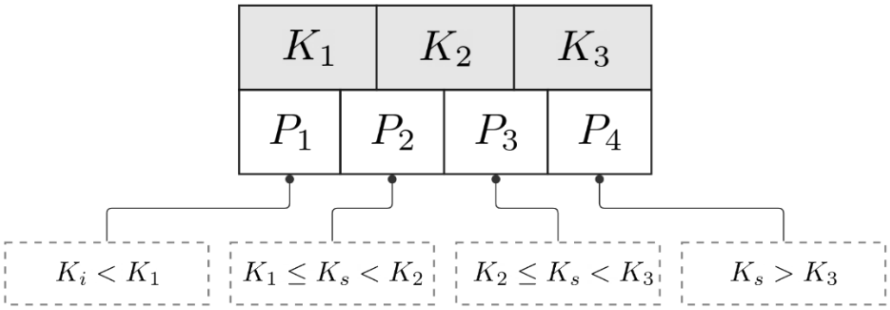

Рисунок 1.2. Структура внутрішнього вузла B+-дерева. K – це множина ключів, а K_s – ключ.

Процес пошуку потрібного запису завжди розпочинається з кореня і проходить деревом униз, порівнюючи шукане значення з ключами у відповідних вузлах. Складність пошуку в B+-дереві можна оцінювати двома способами: або за кількістю операцій читання сторінок, або за кількість порівнянь, що виконуються під час пошуку [4, с. 37].

Якщо рахувати операції читання, тоді складність пошуку становить $log_K M$, де K – це степінь розгалуження дерева, а M – загальна кількість елементів у дереві. З іншого боку, пошук у середині вузла здійснюється за допомогою бінарного пошуку, тому кількість порівнянь, яку потрібно здійснити, дорівнює $log_2 M$. Це означає, що для пошуку у дереві індексу зі 100 мільйонами записів і степенем розгалуження 100, доведеться здійснити лише чотири операції читання з диску.

Застосування B+-дерев дає змогу ефективно виконувати як точкові запити, так і запити на діапазон значень. У другому випадку складність пошуку дорівнюватиме довжині шляху від кореня до листків плюс кількість листкових сторінок, значення яких входять у діапазон.

Попри значне зменшення кількості необхідних операцій читання та хорошу оптимізацію виконання запитів на пошук, у B+-дереві потрібно також враховувати складність операцій оновлення даних. Під час вставки або видалення запису з B+-дерева може виникнути необхідність в операції розділення або злиття вузлів. При цьому дерево має залишатися збалансованим.

Варто зазначити, що в літературі та документації різних СКБД існує розбіжність у чіткому визначенні різниці між B+-деревами та B-деревами. B-дерева можуть містити дані або указники на будь-якому рівні: у кореневому, внутрішніх та листкових вузлах. Водночас, B+-дерево зберігає дані або указники на них винятково у листкових вузлах. Внутрішні ж вузли B+-дерева, як було зазначено, містять лише ключі, які спрямовують пошук до потрібного значення, розміщеного у листках. Отож кожна операція у B+-дереві (вставка, оновлення, видалення та читання) зачіпає завжди лише листкові вузли, а вищі рівні зазнають модифікації лише у випадку розділення або злиття вузлів [4, с. 36].

1.3. Індекс на основі хеш-таблиці

В окремих ситуаціях альтернативою B+-дереву у якості структури індексу можуть бути хеш-таблиці. У такому випадку індекс ділиться на сегменти, кожен з яких може містити один або більше записів. Самі ж сегменти складаються зі сторінок. За значенням ключа хеш-функція обчислює адресу сегменту, у якому розмістити відповідний запис. Для знаходження потрібного запису необхідно обчислити значення хеш-функції для його ключа та перейти до сегменту з отриманою адресою.

Утім, різні значення ключів можуть отримати однакове хеш-значення. У цій ситуації один сегмент міститиме більше ніж один запис, тож виникає необхідність послідовно перевірити всі елементи сегменту для виявлення шуканого.

Якщо у сторінці сегменту не залишається вільного місця для нових записів, тоді до неї приєднується так званий сегмент переповнення [3, с. 650].

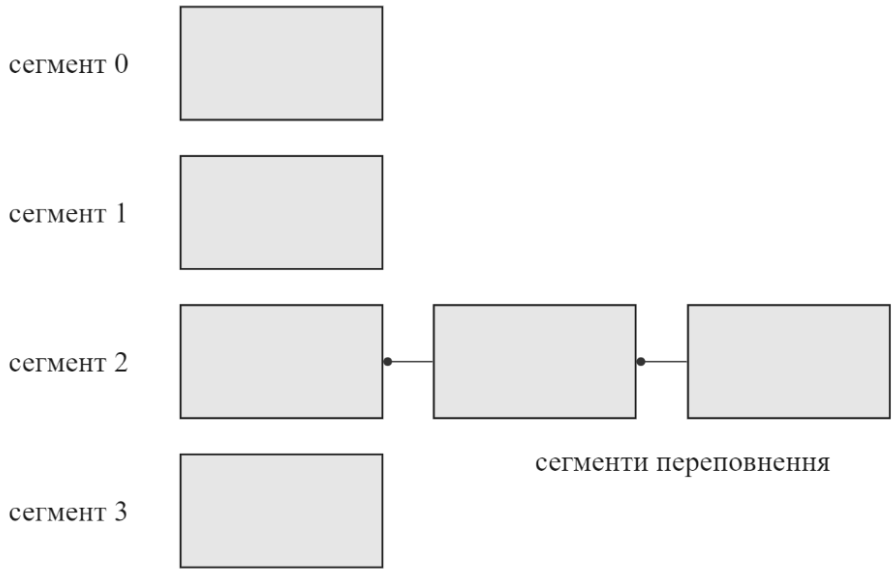

Рисунок 1.3. Структура хеш-індексу

Бажано, щоби кожен сегмент розміщувався в одному блоці, тоді процедура пошуку вимагає виконання лише одної дискової операції введення-виведення. Але в міру збільшення даних у файлі виникає ситуація, коли для збереження вмісту сегменту доводиться використовувати досить довгий ланцюжок сторінок [3, с. 651]. Тоді під час пошуку здійснюватиметься стільки читань з диску, скільки й сторінок займає один сегмент.

На відміну від B+-дерев, хеш-індекси не підтримують запити на діапазон. Вони дають змогу ефективно виконувати лише запити на рівність.

Розділ 2. Реалізація індексів у PostgreSQL

2.1. Короткий огляд PostgreSQL

PostgreSQL є об’єктно-реляційною системою керування базами даних із відкритим кодом, початково розробленою в Університеті Каліфорнії, Берклі. PostgreSQL написана мовою C, має клієнт-серверну архітектуру, підтримує платформи Linux, macOS, Microsoft Windows та ін. Останньою версією на сьогодні є PostgreSQL 15.

PostgreSQL забезпечує паралельний доступ за допомогою MVCC, надає потужні механізми транзакцій і реплікації, повністю відповідає властивостям ACID, підтримує представлення, послідовності, успадкування, зовнішні з’єднання, підзапити, користувацькі функції та ін.

2.2. Типи індексів у PostgreSQL

2.2.1. Типи індексів за структурою

PostgreSQL підтримує чи не найбільшу кількість різновидів індексів серед усіх СКБД, хоча стандартним і найчастіше вживаним є індекс на основі B+-дерева (названий B-Tree), який створюється за замовчуванням командою CREATE INDEX.

B-Tree індекс

B-Tree індекс у PostgreSQL – це дерево з багаторівневою структурою, у якій кожен рівень може бути використаний як двобічно зв’язаний список сторінок. Листкові сторінки складаються з кортежів (tuples), які містять посилання на фізичне розміщення рядків таблиці в кучі з даними, а сторінки внутрішніх вузлів – посилання на нижчі рівні дерева [5].

Індекси на основі B+-дерева можуть бути застосовані у PostgreSQL за виконання запитів для знаходження одного або діапазону значень, які задовольняють умову рівності (=) або нерівності (<, <=, >=, >), а також з використанням операторів IN, BETWEEN, IS NULL, IS NOT NULL за значенням ключа пошуку. Оптимізатор PostgreSQL також може вирішити використовувати індекс у запитах з оператором LIKE у разі, якщо шаблон містить фіксований префікс, наприклад LIKE ‘name%’.

Хеш-індекс

Іншим типом індексу в PostgreSQL є хеш-індекс, який містить 4-байтовий хеш-код, обчислений на основі значення індексованого стовпця, і працює лише для умов на рівність, тому його застосування досить обмежене. Водночас, хеш-індекси займають помітно менше простору, ніж B-Tree індекси. До того ж їх можна побудувати для значень, для яких неможливо визначити лінійний порядок.

У попередніх версіях PostgreSQL використання хеш-індексів було не рекомендованим через низку наявних обмежень [6], які пізніше були виправлені, тож у поточній версії жодних застережень з цього приводу немає [5].

GiST та SP-GiST індекси

PostgreSQL також підтримує індекс типу GiST (Generalized Search Tree), який, як і B-Tree індекс, використовує структуру збалансованого дерева пошуку, але, водночас, є шаблоном для реалізації інших методів індексування. Перевагою цього підходу є надання інтерфейсу для створення нових типів індексів, специфічних для конкретної предметної області, наприклад, геометричних даних або геоданих, без необхідності розуміння деталей внутрішньої реалізації бази даних.

Іншим узагальненим деревом пошуку в PostgreSQL є SP-GiST, що можуть працювати в таких самих ситуаціях як GiST, але швидше для певних типів розподілів даних [7, с. 156].

GIN індекс

Індекс GIN (Generalized Inverted Index) у PostgreSQL застосовується у випадках, коли потрібно індексувати складені значення, і розрахований на пошук складових елементів у цих значеннях. GIN індекс також має структуру B+-дерева, побудованого на основі ключів, які є елементами індексованих складених об’єктів. На рівні листків кожен кортеж сторінки містить або указник на дерево указників на дані, або список указників на дані, якщо він має достатньо малий розмір [5]. Основною сферою застосування GIN індексу є повнотекстовий пошук.

BRIN індекси

BRIN індекс (Block Range Index) використовується у PostgreSQL для таблиць дуже великого обсягу, для яких індекс на основі B+-дерева не зможе повністю розміститися в оперативній пам’яті і у яких значення певних стовпців корелюють з розташуванням рядків на фізичному рівні. Для кожної групи сторінок, які розміщуються поряд, тобто зони блоків, BRIN індекс містить певну загальну інформацію, найчастіше це найменше і найбільше значення. Під час виконання запиту це допомагає пропускати окремі зони блоків, які не потрапляють у вказаний діапазон, і в такий спосіб пришвидшити послідовне сканування таблиці [5].

2.2.2. Типи індексів за застосуванням

Складені індекси

PostgreSQL підтримує створення B-Tree, GiST, GIN, BRIN індексів для таблиць одразу на основі декількох ключів пошуку, при цьому максимальна кількість стовпців в індексі становить 32, включно з доданими за допомогою команди INCLUDE [5], яка у PostgreSQL дає змогу розміщувати в індексі стовпці з даними, не індексуючи їх, що може бути корисно для використання цього індексу як покриваючого.

Унікальні індекси

Особливий різновид індексів у PostgreSQL – унікальні індекси (unique index). Вони будуються на основі B-Tree індексу та створюються автоматично під час накладання обмеження на унікальність звичайного стовпця або первинного ключа за створення таблиці. Тож для унікальних стовпців явно створювати індекси не потрібно.

Індекси виразів

Індекси можна створювати на лише за значеннями стовпців, але за результатом обчислення виразів над одним або декількома стовпцями. Утім підтримання такого індексу вимагає значних витрат, адже обчислення повинні здійснюватися щоразу, коли додається або оновлюється рядок.

Часткові індекси

У PostgreSQL надається можливість створювати індекси для підмножини рядків таблиці, для яких виконується певна умова. Такий індекс називають частковим (partial index). Виключення зайвих рядків допомагає пришвидшити не лише виконання запитів на читання, але й на оновлення даних, адже потреба застосовувати зміни до індексу виникатиме рідше.

Покриваючі індекси

Зазвичай під час виконання запиту з використанням індексу потрібно зчитувати записи спершу з індексу, а потім з таблиці даних. Однак, якщо в індексі вже наявні всі потрібні запиту значення, оптимізатор PostgreSQL може вирішити застосувати сканування лише індексу (Index Only Scan). У такому випадку індекс називатиметься покриваючим (covering) для цього запиту. Щоб зробити індекс покриваючим, PostgreSQL надає можливість розміщувати у ньому стовпці без індексування за допомогою ключового слова INCLUDE.

2.3. Застосування індексів у PostgreSQL

Індекси в PostgreSQL можуть використовуватися для оптимізації виконання запитів SELECT, UPDATE, DELETE з умовою WHERE, а також для операції JOIN.

Вказівку ORDER BY оптимізатор виконує здійснюючи явне сортування або за допомогою B-Tree індексу. Однак за потреби опрацювання значної частини таблиці, явне сортування все ж виконуватиметься швидше, адже для зчитування даних використовуватиметься послідовний доступ.

Створення індексу зазвичай займає тривалий час. За замовчуванням PostgreSQL може виконувати побудову індексу лише паралельно з виконанням команди SELECT, блокуючи запити на запис до таблиці. Утім, команда CREATE INDEX CONCURRENTLY надає можливість створення індексу без блокування запису, але виконується значно довше й потребує більше ресурсів, аніж звичайна процедура побудови індексу [5].

2.3.1. Читання даних за допомогою індексу

У PostgreSQL немає первинних індексів, тобто всі індекси завжди зберігаються за межами таблиць із даними і містять посилання на фізичні адреси рядків (tuple id) у кучі. Під час сканування індексу це дає змогу безпосередньо вибирати потрібні рядки з таблиці.

На відміну від індексу, в якому елементи, що відповідають умові запиту, розміщуються поряд, дані в таблиці розташовуються у довільний спосіб, адже в PostgreSQL рядки таблиці зберігаються у кучі, тобто невпорядковано. Це означає, що звернення до таблиці з даними під час пошуку з використанням індексу призводить до низки довільних операцій введення-виведення, які здійснюються повільніше за послідовне читання. Отож, якщо оптимізатор PostgreSQL виявить, що зчитати необхідно лише невелику кількість рядків, тоді він застосує індекс, в

іншому разі, навіть попри наявність індексу, він використовуватиме послідовне сканування всієї таблиці.

Окрім послідовного сканування та сканування індексу, оптимізатор PostgreSQL також може вирішити виконувати запит у два етапи: спершу переглянути індекс і позначити адреси потрібних рядків у бітовій карті, а потім за допомогою цієї карти зчитувати дані з кучі. Такий метод доступу називається скануванням бітової карти. У випадку потреби опрацювати великий обсяг даних він працює ефективніше за звичайне сканування індексу, адже уникає повторного звернення до однієї і тієї ж сторінки [8].

2.3.2. Оновлення даних за допомогою індексу

B-Tree індекс у PostgreSQL не враховує, що механізм MVCC допускає наявність одразу декількох версій одного рядка в таблиці, тому вважає кожен кортеж окремим об’єктом [5]. Через це може відбуватися накопичення “відпрацьованих” кортежів (неактуальних версій рядка). До того ж під час виконання команди UPDATE для зміни значення навіть одного індексованого стовпця здійснюється додавання нового кортежу (останньої версії рядка) до всіх наявних для відповідної таблиці індексів. Однак, необхідність під час виконання запиту в розділенні вузла дерева може спричинити висхідне видалення індексних кортежів, тобто очищення сміття в індексі. Наявність значної кількості різних версій одного рядка є важливим негативним фактором загальної ефективності системи [5].

З метою запобігти значних витрат під час оновлення у PostgreSQL наявний механізм оптимізації, який називається “кортежі лише в кучі” (Heap-Only Tuples) [5]. Він застосовується, якщо жоден індексований стовпець не задіяний в операції UPDATE.

Розділ 3. Реалізація індексів у MySQL

3.1. Короткий огляд MySQL

Згідно з офіційною документацією, MySQL, найбільш популярна реляційна система керування базами даних, розроблена та підтримувана корпорацією Oracle [9]. MySQL написана мовами C та C++, працює з низкою різних платформ (Linux, macOS, Microsoft Windows та ін.), використовує багаторівневу серверну архітектуру. Останньою версією на сьогодні є MySQL 8.0.

MySQL підтримує різні типи рушіїв (InnoDB, MyISAM, MEMORY, ARCHIVE та ін.), які призначені для забезпечення виконання різних вимог щодо підтримки транзакцій, способів збереження даних, швидкості виконання окремих операцій, стратегій забезпечення паралельного доступу тощо. У MySQL 8.0 за замовчуванням використовується InnoDB, рушій загального призначення, який поєднує високу надійність та ефективність [9].

Оскільки індекси в MySQL реалізовані на рівні рушія, а не серверу, вони не є стандартизованими: їхні типи та принципи роботи відрізняються у різних рушіях [10, с. 156]. Отож далі буде розглянуто особливості імплементації індексів саме для InnoDB.

3.2. Типи індексів у MySQL

3.2.1. Типи індексів за структурою

За замовчуванням під час створення таблиці MySQL будує кластерний індекс на основі B+-дерева (B-tree в документації), внутрішні вузли якого містять указники на нижчі рівні дерева, а в листках безпосередньо розміщуються дані таблиці. Кожен рівень сторінок формує двобічний зв’язаний список. У межах сторінки записи не впорядковані фізично: вони займають місце, яке доступне під час вставки [11]. На

основі B+-дерев MySQL будує індекси PRIMARY KEY, UNIQUE, INDEX і FULLTEXT.

B-tree індекси в MySQL застосовується для пошуку одного значення або діапазону значень, для яких виконується умова на рівність (=), нерівність (<, <=, >=, >) або оператори BETWEEN, IN. Також він може бути застосований у запитах, що містять умову на збіг постійного префіксу в текстовому значенні за використання оператора LIKE.

InnoDB не підтримує явного створення індексу на основі хеш-таблиці, але може використовувати адаптивний хеш-індекс, механізм якого здатен виявляти, що до певних значень індексу доступ здійснюється досить часто, і тому автоматично будує для них хеш-індекс в оперативній пам’яті на основі B+-дерева [10, с. 159].

Для індексування просторових даних InnoDB використовує R-дерево, яке є багатовимірним розширенням одновимірного B+-дерева. Структуру R-дерева мають індекси типу SPATIAL.

3.2.2. Типи індексів за застосуванням

Кластерний індекс

За створення таблиці InnoDB автоматично будує кластерний індекс за первинним ключем на основі B+-дерева. Такий індекс має назву PRIMARY KEY. Якщо для таблиці не було зазначено первинного ключа, а також на жоден з її стовпців не було накладено обмеження на унікальність, InnoDB згенерує неявний кластерний індекс на основі фізичного ідентифікатора рядка [9].

Унікальний індекс

UNIQUE індекси використовуються для гарантування унікальності значення стовпця. На відміну від індексу PRIMARY KEY, одна таблиця може мати декілька UNIQUE індексів.

Складені індекси

MySQL підтримує створення індексів за декількома стовпцями. Щонайбільше складений індекс може мати 16 ключів. Такий індекс допомагає пришвидшувати пошук за значенням першого стовпця, перших двох стовпців, перши трьох стовпців і т.д.

Покриваючі індекси

Якщо в індексі розміщуються всі стовпці, які потрібно отримати, тоді він називатиметься покриваючим для цього запиту, адже оптимізатор MySQL не матиме потреби додатково читати дані з таблиці.

3.3. Застосування індексів у MySQL

MySQL використовує B+-індекси для

- швидкого знаходження рядків, які задовольняють умову WHERE;
- отримання рядків з інших таблиць під час виконання операції JOIN;
- знаходження значень функцій MIN() і MAX();
- сортування або групування таблиці;
- отримання значень (у випадку покриваючого індексу).

3.3.1. Читання даних за допомогою індексу

InnoDB має два різновиди індексів: первинні (кластерні) та вторинні. Як було зазначено, дані таблиць зберігаються не окремо, а безпосередньо в кластерних B+-індексах. Застосування кластерних індексів в InnoDB має важливі переваги, які дають можливість покращити ефективність виконання запитів, а саме

- розміщення записів з близькими значеннями поряд (або послідовно), що запобігає необхідності виконувати операції довільного введення-виведення;

- швидкий доступ до даних без потреби окремо звертатися до індексу, а потім до основної таблиці.

Водночас, індекси, які створюються явно, є вторинними. Кожен запис у вторинному індексі в InnoDB містить значення ключа пошуку, а також значення первинного ключа, яке використовується як посилання на потрібний запис у кластерному індексі з даними. Тож для отримання значень під час сканування вторинних індексів потрібно також звертатися й до первинного індексу, а потім зчитувати дані.

3.3.2. Оновлення даних в MySQL

У MySQL записи вторинних індексів містять значення первинного ключа, а не фізичної адреси рядка. Оскільки самі дані зберігається у листках кластерного індексу, оновлення запису відбувається одразу на місці й індекси, значення ключів яких не були оновленні, залишаються незмінними.

З цього також випливає, що оновлення значення первинного ключа в MySQL є надзвичайно дорогою операцією, хоча й потреби у її виконанні в коректно спроектованій базі даних не має виникнути.

Розділ 4. Відмінності індексів у PostgreSQL і MySQL

4.1. Порівняння ефективності читання та оновлення даних

Читання даних у загальному випадку відбувається ефективніше у PostgreSQL, адже кожен індекс безпосередньо містить значення фізичних адрес рядків у кучі. В InnoDB за використання вторинних індексів здійснюється дворівнева адресація: спершу сканується вторинний індекс для знаходження значення первинного ключа потрібного рядка, потім відбувається звернення до первинного індексу для отримання даних.

Натомість, оновлення даних у MySQL відбувається швидше: зміни застосовуються лише до того індексу, стовпець якого зазнав оновлення. У PostgreSQL запит на оновлення може призвести до такого явища, як “посилення запису” (write amplification), яке полягає в тому, що за незначних змін логічного рівня виконуються досить витратні операції на фізичному рівні. Цей недолік свого часу став однією з причин міграції сервісу Uber з PostgreSQL на MySQL [12]. Наприклад, оновлення одного значення в записі таблиці призводить до необхідності виконувати витратні дискові операції запису для всіх створених для неї індексів, навіть тих, які не містять значення, яке має бути оновлене. Це помітно сповільнює виконання запиту.

4.2. Зіставлення основних можливостей та розширень

У розділах 2 і 3 було детально розглянуто різновиди індексів та їхнє застосування у PostgreSQL і MySQL. Підсумок основних відмінностей у можливостях та розширеннях, що надаються кожною з розглянутих СКБД, представимо у вигляді порівняльних таблиць.

З Таблиці 4.1, у якій наведено порівняння наявних типів індексів з огляду на їхнє застосування, можна зробити висновок, що PostgreSQL має підтримку більшої кількості різновидів індексів, що дає помітну гнучкість в оптимізації запитів. До того ж у PostgreSQL є додаткова можливість розміщувати окремі стовпці, не індексуючи їх, що розширює застосування покриваючого індексу. Водночас, у MySQL дані автоматично розміщуються в первинних індексах.

Таблиця 4.1. Типи індексів за застосуванням

	PostgreSQL	**MySQL (InnoDB)**
Складений індекс	так	так
Унікальний індекс	так	так
Індекс виразу	так	ні
Частковий індекс	так	ні
Покриваючий індекс	так	так

Порівняння структур, які використовуються для побудови індексів, подано в Таблиці 4.2. PostgreSQL підтримує декілька різних типів індексів для просторових даних, повнотекстового пошуку та інших особливих типів даних, специфічних для певної предметної області, а також надає можливість створювати нові індекси на основі цих структур.

Натомість, у MySQL для просторових даних використовуються лише індекси на основі R-дерева, а для повнотекстового пошуку – B+-дерева. Також MySQL має особливий механізм автоматичної побудови адаптивного хеш індексу. Можливість явного створення хеш індексу в рушії InnoDB відсутня.

Таблиця 4.2. Типи індексів за структурою

	PostgreSQL	**MySQL (InnoDB)**
B+-дерево	так	так
Хеш	так	лише адаптивний
R-дерево	ні	так
GiST	так	ні
SP-GiST	так	ні
GIN	так	ні
BRIN	так	ні

Додаткові можливості та розширення індексів зіставлені в Таблиці 4.3. Наявність у PostgreSQL методу доступу за допомогою створення та сканування бітової карти надає спосіб в окремих випадках уникнути багаторазового довільного читання однієї сторінки в результаті сканування індексу, пришвидшуючи виконання запиту. Як було зазначено раніше, PostgreSQL також надає більш гнучкі механізми для індексування просторових даних.

Таблиця 4.3. Можливості та розширення індексів

	PostgreSQL	**MySQL (InnoDB)**
Повнотекстовий пошук	так	так
Просторові індекси	так	так
Створення нових індексів	так	ні
Сканування бітової карти	так	ні

Розділ 5. Проектна реалізація

5.1. Проектування бази даних

На основі реляційної моделі, наведеної на Рисунку 5.1, у PostgreSQL 15 і MySQL 8 було створено дві бази даних, які містять інформацію про замовлення клієнтів. У результаті заповнення згенерованим (ідентичним для обох СКБД) набором даних таблиця clients містить 80 000 рядків, products – 100 000 рядків, orders – 2 000 000 рядків, order_items – 3 100 000 рядків.

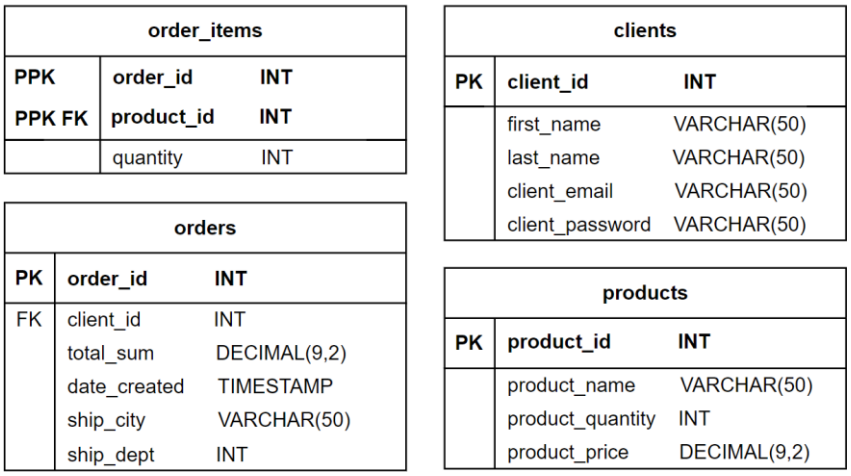

Рисунок 5.1. Реляційна модель

Для усіх таблиць у PostgreSQL автоматично створено вторинні B-tree індекси для стовпців первинних ключів, у MySQL – PRIMARY індекси.

Додаткова визначимо вторинні індекси:

- для таблиці orders:

```
CREATE INDEX i_o_fk_client ON orders (client_id);
CREATE INDEX i_o_date ON orders (date_created);
CREATE INDEX i_o_ship ON orders (ship_city, ship_dept);
CREATE INDEX i_o_total ON orders (total_sum);
```

- для таблиці order_items:

```
CREATE INDEX i_fk_product_id ON order_items (product_id);
```

5.2. Проектування застосунку

Для наочного дослідження та порівняння роботи індексів було розроблено вебзастосунок, який використовує створені у попередньому підрозділі бази даних та дає змогу автоматизовано виконувати вибірку інформації з них, надаючи відомості про план виконання кожного запиту.

Серверна частина застосунку розроблена за допомогою платформи Node.js, мови програмування TypeScript та фреймворку NestJS. Клієнтська частина застосунку розроблена мовою JavaScript з використанням бібліотеки React.

Інтерфейс користувача застосунку наведено на Рисунку 5.1.

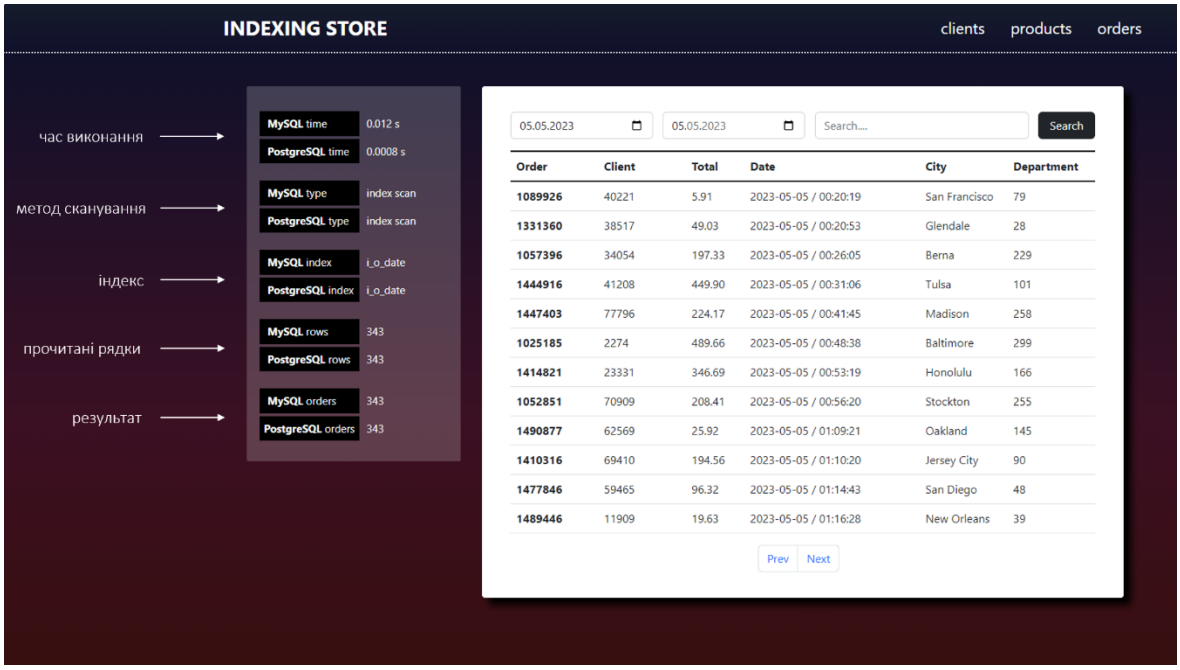

Рисунок 5.1. Інтерфейс користувача

5.3. Аналіз виконання запитів

З метою порівняти роботу індексів окремо розглянемо деякі запити, використані у застосунку. Щоб отримати детальну інформацію про план виконання

запитів і статистику, використаємо команду EXPLAIN ANALYZE. У різних СКБД результат її виконання дещо відрізняється, тому для зручності читання дані будуть представлені у відформатованому варіанті.

Параметри, на які варто звертати увагу: метод доступу (повне сканування таблиці, сканування індексу, сканування бітової карти і т.д.), час виконання, кількість переглянутих сторінок і кількість отриманих сторінок.

Варто зазначити, що ці показники не є ідеальним способом для оцінки ефективності виконання запиту. Зокрема, час виконання одного й того ж запиту може значно відрізнятися, адже на нього впливає низка факторів: навантаження системи, блокування, висока конкурентність, особливості апаратного забезпечення тощо. Натомість, кількість переглянутих сторінок дає розуміння, наскільки ефективно відбувається пошук потрібних даних. В ідеалі кількість переглянутих сторінок має збігатися з кількістю повернених, але на практиці це трапляється нечасто [10, с. 194-195].

Запит 1

Отримаємо інформацію про всі замовлення, зроблені з початку 2010 року.

```
EXPLAIN ANALYZE
SELECT *
FROM orders
WHERE date_created > '2010-01-01 00:00:00';
```

Результат виконання PostgreSQL:

```
Seq Scan on orders (actual time=0.056..249.152 rows=1234986 loops=1)
    Filter: (date_created > '2010-01-01 00:00:00')
    Rows Removed by Filter: 765014
```

Результат виконання MySQL:

```
Filter: (orders.date_created > '2010-01-01 00:00:00')
(actual time=0.058..1532.223 rows=1234986 loops=1)
    Table scan on orders (actual time=0.056..1372.779 rows=2000000 loops=1)
```

Оскільки запит вимагає читання великого обсягу даних, в обох випадках оптимізатор вирішує не застосовувати наявний індекс, а здійснити послідовне сканування всієї таблиці orders. Для цього відбувається читання наявних 2 000 000 рядків і фільтрація, щоб дістати 1 234 986 тих, які задовольняють умову WHERE. Застосування індексу в цьому випадку було б неефективним з огляду на велику кількість операцій довільного доступу, які потрібно було б здійснити.

Запит 2

Отримаємо інформацію про всі замовлення, зроблені 10 жовтня 2018 року.

```
EXPLAIN ANALYZE
SELECT *
FROM orders
WHERE date_created BETWEEN '2018-10-10 00:00:00' AND '2018-10-10 23:59:59';
```

Результат виконання PostgreSQL:

```
Index Scan using i_o_date on orders
(actual time=0.018..0.134 rows=220 loops=1)
    Index Cond: ((date_created >= '2018-10-10 00:00:00') AND
    (date_created <= '2018-10-10 23:59:59'))
```

Результат виконання MySQL:

```
Index range scan on orders using i_o_date over ('2018-10-10 00:00:00' <=
date_created <= '2018-10-10 23:59:59'), with index condition:
(orders.date_created between '2018-10-10 00:00:00' and '2018-10-10
23:59:59') (actual time=0.036..1.954 rows=220 loops=1)
```

В обох випадках оптимізатор вирішує сканувати індекс i_o_date, адже обсяг даних, які потрібно зчитати, значно менший у порівнянні з попереднім запитом і виконання довільного читання є виправданим. І PostgreSQL, і MySQL за допомогою індексу зчитують 220 рядків, які задовольняють умові WHERE.

Запит 3

Отримаємо інформацію про всі замовлення, у яких зазначено місто Токіо.

```
EXPLAIN ANALYZE
SELECT *
```

```
FROM orders
WHERE ship_city = 'Tokyo';
```

Результат виконання PostgreSQL:

```
Bitmap Heap Scan on orders (actual time=3.419..15.437 rows=18680 loops=1)
    Recheck Cond: ((ship_city) = 'Tokyo')
    Heap Blocks: exact=11350
        Bitmap Index Scan on i_o_ship (actual time=2.014..2.015 rows=18680
        loops=1)
        Index Cond: ((ship_city) = 'Tokyo')
```

Результат виконання MySQL:

```
Index lookup on orders using i_o_ship (ship_city='Tokyo')
(actual time=0.865..261.962 rows=18680 loops=1)
```

Оптимізатор PostgreSQL вирішує застосувати сканування бітової карти за індексом i_o_ship, що дає змоги уникнути багаторазових читань однієї й тієї ж сторінки, як це відбувалося б за використання лише індексу, і, водночас, не здійснює сканування всієї таблиці, що також було б неефективно. Натомість, MySQL застосовує індекс для пошуку потрібних 18680 рядків, виконуючи, ймовірно, зайвий раз довільний доступ до однієї і тієї ж сторінки.

Запит 4

Отримаємо інформацію про всі замовлення, у яких були товари з індексами 6700, 6701, 6702.

```
EXPLAIN ANALYZE
SELECT *
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
WHERE product_id IN (6700, 6701, 6702);
```

Результат виконання PostgreSQL:

```
Nested Loop (actual time=0.033..0.363 rows=87 loops=1)
    Bitmap Heap Scan on order_items oi (actual time=0.026..0.090 rows=87
    loops=1)
    Recheck Cond: (product_id = ANY ('{6700,6701,6702}'))
    Heap Blocks: exact=87
        Bitmap Index Scan on i_fk_product_id (actual time=0.014..0.015
        rows=87 loops=1)
        Index Cond: (product_id = ANY ('{6700,6701,6702}'))
```

```
    Index Scan using i_o_pkey on orders o (actual time=0.003..0.003 rows=1
    loops=87)
    Index Cond: (order_id = oi.order_id)
```

Результат виконання MySQL:

```
Nested loop inner join (actual time=0.124..3.980 rows=87 loops=1)
     Index range scan on oi using i_fk_product_id,
     with index condition: (oi.product_id in (6700,6701,6702))
     (actual time=0.106..0.568 rows=87 loops=1)
         Single-row index lookup on o using PRIMARY (order_id=oi.order_id)
         (actual time=0.039..0.039 rows=1 loops=87)
```

У PostgreSQL спершу здійснюється сканування індексу i_fk_product_id на основі первинного ключа product_id і створення бітової карти, за допомогою якої потім відбувається сканування кучі з даними order_items й отримання 87 рядків з потрібними товарами. Сканування індексу i_o_pkey таблиці orders у циклі повертає 87 рядків. MySQL так само здійснює сканування індексу i_fk_product_id для отримання 87 рядків і стільки ж повторних однорядкових сканувань первинного індексу таблиці orders.

Запит 5

Отримаємо інформацію про всі замовлення, у яких зазначено місто Лондон і номер відділення більший за 100, зроблені протягом 2020 року.

```
EXPLAIN ANALYZE
SELECT *
FROM orders
WHERE ship_city = 'London' AND ship_dept > 100 AND date_created BETWEEN
'2020-01-01 00:00:00' AND '2020-12-31 23:59:59';
```

Результат виконання PostgreSQL:

```
Bitmap Heap Scan on orders (actual time=9.075..9.407 rows=561 loops=1)
Heap Blocks: exact=536
    BitmapAnd (actual time=8.959..8.960 rows=0 loops=1)
        Bitmap Index Scan on i_o_ship (actual time=1.381..1.381 rows=11080
        loops=1)
        Index Cond: (((ship_city) = 'London') AND (ship_dept > 100))
        Bitmap Index Scan on i_o_date (actual time=7.124..7.124 rows=111452
        loops=1)
```

```
        Index Cond: ((date_created >= '2020-01-01 00:00:00' AND
        (date_created <= '2020-12-31 23:59:59'))
```

Результат виконання MySQL:

```
Filter: (orders.date_created between '2020-01-01 00:00:00' and '2020-12-31
23:59:59') (actual time=0.043..281.534 rows=561 loops=1)
     Index range scan on orders using i_o_ship over (ship_city = 'London'
     AND 100 < ship_dept), with index condition: ((orders.ship_city =
     'London') and (orders.ship_dept > 100)) (actual time=0.040..271.627
     rows=11080 loops=1)
```

PostgreSQL виконує сканування двох індексів i_o_date і i_o_ship, створюючи бітову карту, в першому випадку переглянувши 111452 рядків, у другому – 11080. У результаті сканування кучі отримує потрібний 561 рядок. MySQL здійснює сканування складеного індексу i_o_ship, отримуючи 11080 рядків, серед яких фільтрує ті, які мають потрібну дату.

Висновки

У курсовій роботі було здійснено дослідження та порівняння особливостей реалізації та застосування індексів у PostgreSQL і MySQL. Виявлені відмінності засвідчили безпосередній вплив різних підходів в архітектурі системи на ефективність її роботи в оптимізації запитів.

Обидві СКБД підтримують декілька видів індексів для різних цілей і типів даних, зокрема, для просторових даних і повнотекстового пошуку. Утім, PostgreSQL надає більше розширень та можливостей у роботі зі складеними даними, специфічними для певної предметної області.

Окремо варто зазначити різні підходи до способу збереження даних. Записи в індексах PostgreSQL завжди містять указники на фізичні адреси відповідних рядків таблиці, що, попри очевидні переваги для операцій читання даних, може приводити до неефективного виконання операцій оновлення рядків. Натомість, архітектурні рішення, використані для MySQL, дають змогу уникнути цих проблем, що, з іншого боку, призводить до потреби здійснювати дворівневу адресацію для читання даних.

Аналіз планів виконання запитів також продемонстрував, що оптимізатор PostgreSQL забезпечує помітно вищу продуктивність, порівняно з результатами, отриманими для MySQL, що може свідчити про більш потужні механізми, закладені в його реалізацію.

Отже, здійснене дослідження дало змогу виявити як переваги, так і обмеження у механізмах роботи індексів у PostgreSQL і MySQL, на які варто звертати увагу під час проектування інформаційної системи. Загалом, вибір залежить від низки факторів: сценарію використання, складності запитів, специфіки й обсягу даних тощо.

Список джерел та літератури

1. Lahdenmaki T., Leach M. Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al. Wiley-Interscience, 2005. 328 p.
2. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts. 7th ed. McGraw-Hill Education, 2019. 1376 p.
3. Garcia-Molina H., Ullman J., Widom J. Database Systems: The Complete Book. 2nd ed. Pearson, 2008. 1248 p.
4. Petrov A. Database Internals: A Deep Dive into How Distributed Data Systems Work. O'Reilly Media, 2019. 370 p.
5. Документація PostgreSQL 15 [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/15/index.html
6. Документація PostgreSQL 9.6 [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/9.6/indexes-types.html
7. Obe R., Hsu L. PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database. 3rd ed. O'Reilly Media, 2017. 312 p.
8. Rogov E. Queries in PostgreSQL: 4. Index scan [Електронний ресурс] – Режим доступу до ресурсу: https://postgrespro.com/blog/pgsql/5969493
9. Документація MySQL 8 [Електронний ресурс] – Режим доступу до ресурсу: https://dev.mysql.com/doc/refman/8.0/en/
10. Botros S., Tinley J. High Performance MySQL: Proven Strategies for Operating at Scale. 4th ed. O'Reilly Media, 2021. 386 p.
11. Cole J. B+Tree index structures in InnoDB [Електронний ресурс] – Режим доступу до ресурсу: https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/
12. Why Uber Engineering Switched from Postgres to MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://www.uber.com/en-US/blog/postgres-to-mysql-migration/