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

Оптимізація роботи СУБД PostgreSQL: від рівня сервера до рівня запитів

Керівник кваліфікаційної роботи
к.н., ст. викл. Захоженко П.О.
_____ (підпис)
“____” _____ 2024 р.
Виконала студентка Живіцька Є. О.
“____” _____ 2024 р.

Київ 2024

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій ефективність та надійність систем управління базами даних (СУБД) є ключовими факторами успіху будь-якої організації. PostgreSQL, як одна з найпопулярніших відкритих СУБД, активно використовується у різних галузях, починаючи від фінансів і закінчуючи науковими дослідженнями. Однак, як і у випадку з будь-якою складною програмною системою, максимізація продуктивності PostgreSQL вимагає глибокого розуміння її тонкощів і вміння оптимізувати її роботу. Це завдання охоплює широкий спектр методів і підходів, починаючи від тонкого налаштування конфігурації сервера і закінчуючи створенням ефективних SQL-запитів.

Оптимізація роботи PostgreSQL, як на рівні сервера, так і на рівні запитів, дозволяє значно підвищити продуктивність та надійність системи, що є актуальним завданням у контексті зростаючих вимог до обробки даних. Тема роботи є надзвичайно актуальною, оскільки вона спрямована на покращення функціонування СУБД, що безпосередньо впливає на ефективність бізнес-процесів і наукових досліджень.

У цій роботі буде досліджуватися оптимізація системи управління базами даних PostgreSQL, що охоплює широкий спектр методів та підходів на різних рівнях. Такий вибір дозволяє комплексно розглянути питання оптимізації, забезпечуючи всебічне розуміння і виявлення можливостей для покращення продуктивності СУБД.

Метою даного дослідження є розробка та обґрунтування методів оптимізації роботи PostgreSQL, що забезпечують підвищення її продуктивності та надійності. Для досягнення цієї мети поставлені наступні задачі: дослідження та аналіз параметрів конфігурації PostgreSQL сервера, дослідження методів оптимізації SQL-запитів, проведення експериментів для оцінки ефективності запропонованих методів.

Для написання роботи використовувались численні джерела інформації, серед яких авторитетні публікації та експертні матеріали. Така різноманітна колекція ресурсів створює міцну основу для нашого дослідження методів і стратегій оптимізації. Офіційна документація PostgreSQL, яка є основним джерелом для дослідження, слугує безцінним ресурсом, пропонуючи вичерпні вказівки щодо тонкощів архітектури системи, параметрів конфігурації та найкращих практик оптимізації. Це авторитетне джерело забезпечує глибоке розуміння основних механізмів, що впливають на продуктивність PostgreSQL, і закладає основу для ефективних стратегій оптимізації.

Таким чином, вибір теми дослідження обґрунтований її актуальністю та значущістю, а комплексний підхід до вивчення питання дозволяє досягти поставленої мети та вирішити визначені завдання.

1. Підготовка сервера до тестування

1.1.Docker Compose

Docker Compose - це інструмент для визначення та запуску багатоконтейнерних додатків. Він дозволяє керувати набором контейнерів, кожен з яких представляє окремий сервіс проекту. Керування включає збірку, запуск з залежностями та конфігурацію.[1]

Конфігурація Docker Compose описана у файлі `docker-compose.yml`, розташованому в корені проекту. Він буде використовуватися для створення декількох контейнерів з різними конфігураціями PostgreSQL сервера за допомогою postgres зображення. Приклад конфігурації контейнеру:

- postgres1:

```
image: postgres:latest
environment:
  POSTGRES_DB: postgres
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: postgres
ports:
  - "5433:5432"
volumes:
  - ./inits/default:/docker-entrypoint-initdb.d
```

1.2.Схема бази даних

Схема тестованої бази даних буде мати наступний вигляд:

```
CREATE TYPE "room_type" AS ENUM(
    'standard',
```

```
    'superior',
    'luxury'
);

CREATE TABLE "hotels"(
    "id" uuid,
    "name" text NOT NULL,
    "star_number" int NOT NULL,
    "address" jsonb NOT NULL,
    "url" text,
    "description" text,
    "created_at" timestamp NOT NULL DEFAULT NOW(),
    PRIMARY KEY ("id")
);

CREATE TABLE "rooms"(
    "id" uuid,
    "hotel_id" uuid,
    "number" int NOT NULL,
    "name" text,
    "price" int NOT NULL,
    "floor" int NOT NULL,
    "beds_number" int NOT NULL,
    "type" room_type NOT NULL,
    "is_smoking_allowed" boolean NOT NULL,
    "description" text,
    "created_at" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY ("id"),
```

```
    FOREIGN KEY ("hotel_id") REFERENCES "hotels"("id")
);

CREATE TABLE "users"(
    "id" uuid,
    "email" text NOT NULL,
    "phone" text NOT NULL,
    "encrypted_password" text NOT NULL,
    "created_at" timestamp NOT NULL DEFAULT NOW(),
    PRIMARY KEY ("id")
);

CREATE TABLE "bookings"(
    "room_id" uuid,
    "user_id" uuid,
    "date_from" timestamp NOT NULL,
    "date_to" timestamp NOT NULL,
    "people_number" int NOT NULL,
    "comment" text,
    "created_at" timestamp NOT NULL DEFAULT NOW(),
    PRIMARY KEY ("room_id", "user_id"),
    FOREIGN KEY ("room_id") REFERENCES "rooms"("id"),
    FOREIGN KEY ("user_id") REFERENCES "users"("id")
);

CREATE TABLE "profiles"(
    "user_id" uuid,
    "first_name" text NOT NULL,
```

```
    "last_name" text NOT NULL,
    PRIMARY KEY ("user_id"),
    FOREIGN KEY ("user_id") REFERENCES "users"("id")
);

CREATE TABLE "users_hotels"(
    "user_id" uuid,
    "hotel_id" uuid,
    "score" int NOT NULL,
    "comment" text,
    "created_at" timestamp NOT NULL DEFAULT NOW(),
    PRIMARY KEY ("user_id", "hotel_id"),
    FOREIGN KEY ("user_id") REFERENCES "users"("id"),
    FOREIGN KEY ("hotel_id") REFERENCES "hotels"("id")
);
```

1.3.Snaplet

Snaplet – це інструмент для автоматичної генерації даних. Він використовує технологію штучного інтелекту для створення подібних до виробничих даних на основі схеми бази даних, покращуючи модульне тестування, та локальні процеси розробки.[2]

Приклад використання:

```
await seed.hotels(x =>
  x(HOTELS_NUMBER, {
    id: ctx => copycat.uuid(ctx.seed),
    star_number: ctx => copycat.int(ctx.seed, STAR_NUMBER),
    address: ctx => {
```

```
      return {
        street: copycat.streetAddress(ctx.seed),
        zip_code: copycat.int(ctx.seed, { min: 1000, max: 9000 }),
        city: copycat.city(ctx.seed),
        country: copycat.country(ctx.seed),
      }
    }
  })
)
```

Цей код генерує вказану кількість рядків (HOTELS_NUMBER) у таблиці hotels. Також Snaplet дозволяє специфікувати типи даних при потребі.

2. Оптимізація серверу

2.1. work_mem

Параметр конфігурації work_mem - це базовий максимальний обсяг пам'яті, який використовується операцією сортуванням в запиті або хеш-таблицею, перед записом у тимчасові файли на диску.[3]

Підвищення значення work_mem може покращити продуктивність деяких запитів, особливо тих, що вимагають великої кількості сортування або хешування. Однак важливо враховувати, що підвищення цього значення може збільшити використання пам'яті сервером та збільшити його навантаження.

За замовчуванням work_mem дорівнює 4 МБ. Якщо його значення вказано без одиниць виміру, воно розглядається як кілобайти (КБ):

- postgres=# SHOW work_mem;
  work_mem

  4MB
  (1 row)

Важливо зауважити, що складний запит може виконувати кілька операцій сортування і хешування одночасно. Кожна операція, як правило, використовує стільки пам'яті, скільки вказано в цьому значенні, перед тим як почати записувати дані в тимчасові файли. Крім того, такі операції можуть виконуватися одночасно кількома запущеними сеансами. Отже, загальний обсяг використаної пам'яті може багаторазово перевищувати значення work_mem. Цей факт потрібно враховувати при виборі значення.

Наприклад, якщо встановити значення 50 МБ, і 30 користувачів надішлють запити, то незабаром буде використовуватися 1,5 ГБ пам'яті.[4]

Щоб обсяг використовуваної памяті не перевищував очікувану норму, важливо враховувати значення max_connections (параметр, який визначає максимальну кількість одночасних з'єднань).

- postgres=# SHOW max_connections;
  max_connections

  100
  (1 row)

Подивитися кількість підключень до окремої бази даних можна за допомогою наступного запиту:

- postgres=# SELECT datname, numbackends
  FROM pg_stat_database;
  datname | numbackends
  ----------+----------
  | 0
  postgres | 1
  template1 | 0
  template0 | 0
  (4 rows)

Або отримати загальну кількість підключень:

- postgres=# SELECT sum(numbackends) as total_connection_number
  FROM pg_stat_database;
  total_connection_number

  1
  (1 row)

Рекомендації щодо підбору значення для параметру work_mem:

- work_mem = (25 % of RAM) / max_connections [5]

Обсяг оперативної пам’яті можна за допомогою наступної команди:

- # cat /proc/meminfo

  MemTotal: 8029624 kB ≈ **7841 MB**

Отже, в нашому випадку оптимальне значення work_mem дорівнює 20 МБ ≈ (25% * 7841 МБ) / 100

Проведемо тестування з такими конфігураціями:

- work_mem = 4 MB (за замовчуванням)
- work_mem = 10 MB
- work_mem = 30 MB
- work_mem = 60 MB

Проведемо тестування запиту збільшуючи кількость під’єднань:

- SELECT * FROM rooms ORDER BY price ASC

В цьому запиті пам’ять work_mem буде використовуватися для сортування кімнат за ціною.

Максимальна кількість під’єднань: 90.

Кількість рядків в таблиці: 100 000.

Графік (Рисунок 1) чітко демонструє, що збільшення параметра work_mem призводить до значного прискорення виконання запитів сортування. Це значно покращує роботу сервера, але важливо зазначити, що збільшувати значення work_mem слід з обережністю. При великій кількості одночасних підключень надмірне використання оперативної пам'яті може призвести до негативних наслідків.

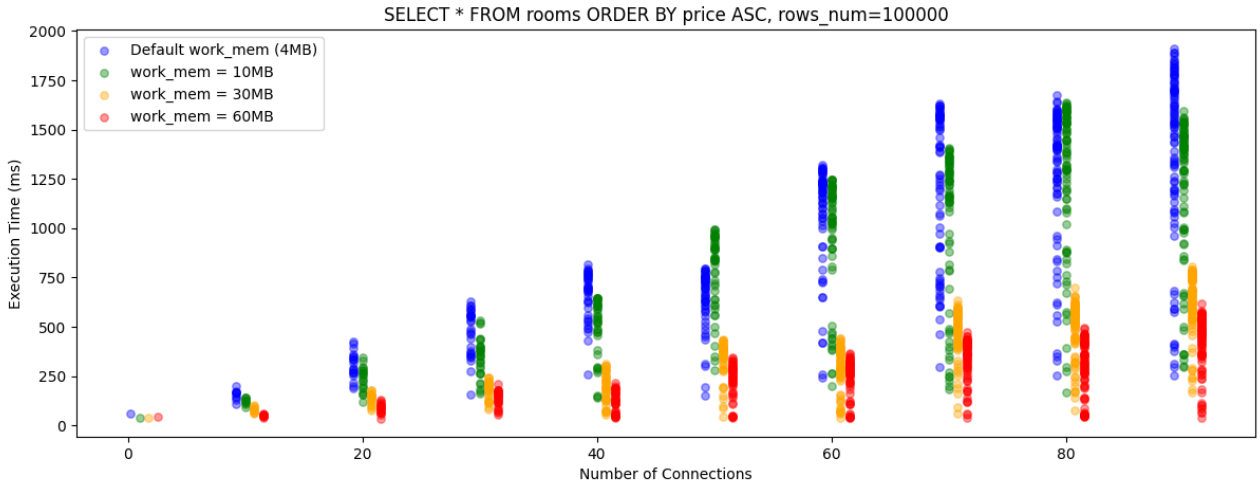

Рисунок 1

Максимальна кількість одночасних під'єднань 90. Очікується, що при максимальній кількості під'єднань максимальне використання пам'яті становитиме:

- 360 МБ (при work_mem = 4 МБ)
- 900 МБ (при work_mem = 10 МБ)
- 2700 МБ (при work_mem = 30 МБ)
- 5400 МБ (при work_mem = 60 МБ)

2.2. shared_buffers

Параметр конфігурації shared_buffers визначає, скільки пам'яті виділяється PostgreSQL для кешування даних. PostgreSQL підтримує кешування як необроблених даних, так і індексів у пам'яті. Дані зберігаються на "сторінках", які по суті є частинами даних, що стосується як файлів даних, так і індексів. Наявність або відсутність сторінки в пам'яті визначає, чи вона кешована.

За замовчуванням shared_buffers дорівнює 128 МБ, але може бути менше, якщо налаштування вашого ядра не підтримують це значення. Цей параметр має бути щонайменше 128 КБ. Однак, для хорошої продуктивності зазвичай потрібні значення, значно більші за мінімальні.[3]

- postgres=# SHOW shared_buffers;
  shared_buffers

  128MB
  (1 row)

Кеш у PostgreSQL тісно співпрацює з кешем операційної системи. Це означає, що коли PostgreSQL шукає дані, він спочатку перевіряє їх у власному кеші. Якщо дані не знайдено у кеші PostgreSQL, далі відбувається пошук у кеші операційної системи. Лише якщо дані не знайдено і там, PostgreSQL звертається до диску для їх читання (Рисунок 2 [6]).

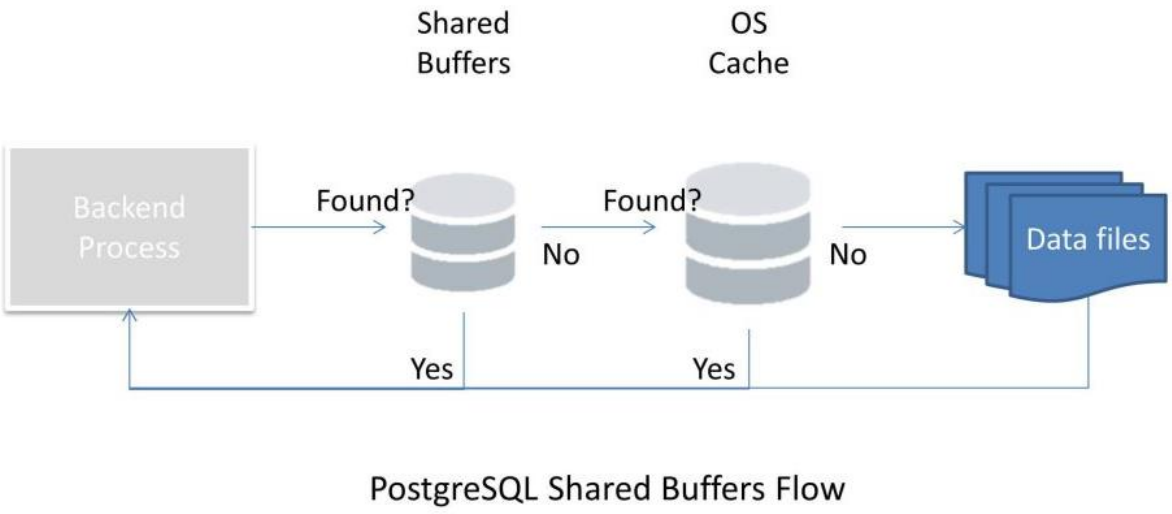

Рисунок 2

Кеш операційної системи є швидким, але доволі простим. Він видаляє старі дані при додавані нових (LRU - витіснення давно невикористовуваних) для виселення кешу, в той час як кеш PostgreSQL зберігає лічильники використання найбільш використовуваних даних, що робить його більш ефективним для повторюваних результатів та індексів.

Хоча кеш shared_buffers у PostgreSQL є більш ефективним, він не може повністю замінити кеш операційної системи. Надмірне збільшення shared_буферів може призвести до зменшення віддачі та помилок, пов'язаних з виходом за межі пам'яті.

Тому його розмір рекомендується встановлювати як помірний відсоток від загальної оперативної пам'яті [6]

Рекомендований розмір shared_buffers зазвичай становить 25% від загального обсягу оперативної пам'яті системи, якщо обсяг оперативної пам’яті більше або дорівнює 1 ГБ. Проте, оптимальне значення може відрізнятися залежно від конкретного навантаження на базу даних.[4]

Перевіритти обсяг оперативної пам’яті можна за допомогою наступної команди:

- # cat /proc/meminfo

  MemTotal: 8029624 kB ≈ **7841 MB**

Отже, в цьому випадку рекомендується використовувати 1960 МБ ≈ 7841 MB * 25%.

Проведемо тестування з такими конфігураціями:

- shared_buffers = 128 MB (за замовчуванням)
- shared_buffers *= 2
- shared_buffers *= 4

Тестований запит (запит виконується 10 разів для кожного випадку):

- SELECT * FROM rooms

Максимальна кількість рядків в таблиці: 1 000 000.

Графік (Рисунок 3) чітко демонстує, що при збільшенні кількості рядків в таблиці більше значення параметру shared_buffers є більш виграшним по часу. При малих обсягах таблиці різниця не є помітною, оскільки кожний сервер використовує один і той самий обсяг буферів.

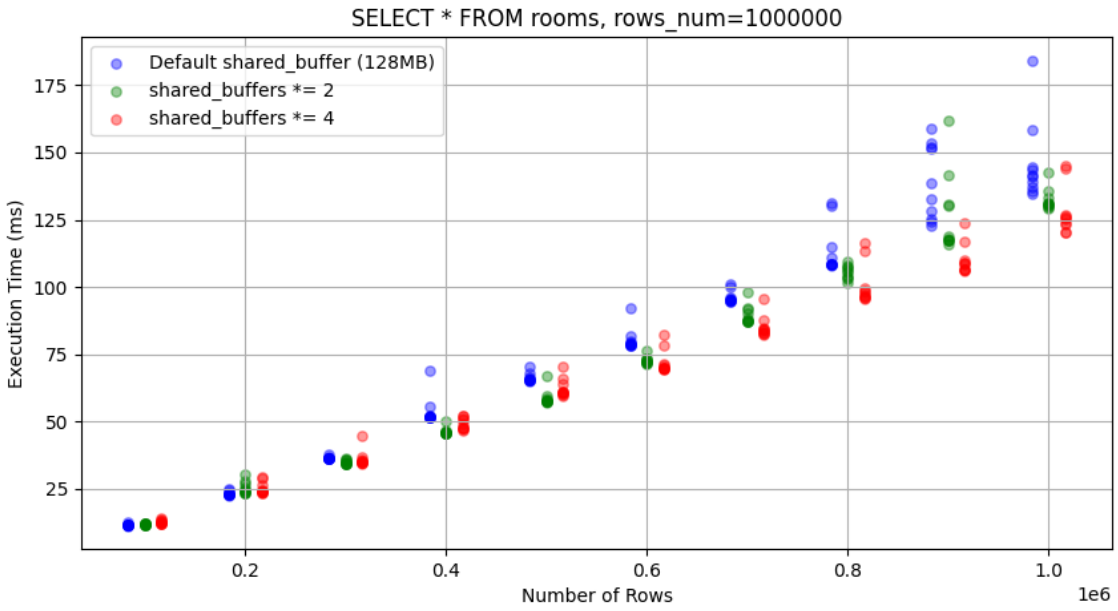

Рисунок 3

Для відслідковування стану shared_buffers можна скористатися розширенням pg_buffercache[6]:

- postgres=# CREATE EXTENSION pg_buffercache;

  CREATE EXTENSION

- postgres=# SELECT * FROM pg_buffercache_summary();

  buffers_used | buffers_unused | buffers_dirty | buffers_pinned | usagecount_avg

  -------------+----------------+---------------+----------------+---------------

  16384 | 0 | 47 | 0 | 0.598388671875

  (1 row)

Розглянемо графік використання shared_buffers (Рисунок 4):

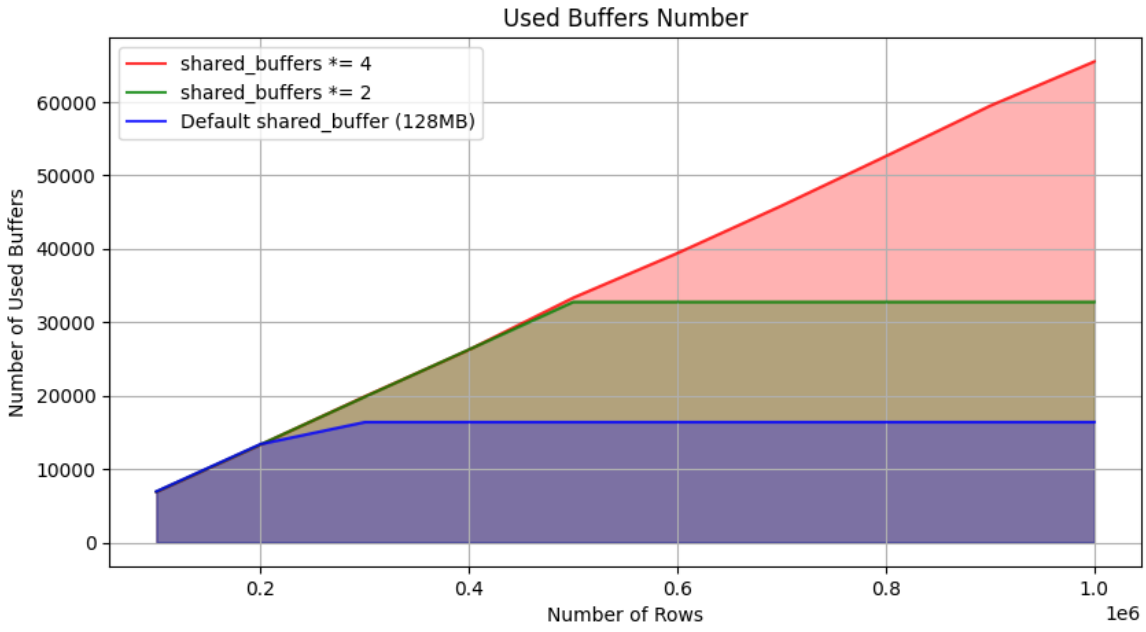

Рисунок 4

Обсяг shared_buffers за замовчуванням переповнився приблизно на 300000 рядках, з подвоєним значенням – на 500000, зі збільшеним вчетверо – не переповнився.

Розмір одного буфера дорівнює розміру блоку. Розмір блок можна перевірити наступним чином:

- postgres=# SHOW block_size;

  block_size

  8192

  (1 row)

Тобто, розмір блоку дорівнює 8192 байти.

2.3. max_parallel_workers_per_gather

Параметр конфігурації max_parallel_workers_per_gather у PostgreSQL встановлює максимальну кількість паралельних робочих процесів, які може використовувати один запит з використанням вузла `Gather` або `Gather Merge`.

За замовчуванням `max_parallel_workers_per_gather` встановлено на 2, це означає, що максимум 2 робочих процеси можуть бути запущені на один паралельний запит. Це обмеження для кожного запиту, на відміну від параметра `max_parallel_workers`, який є загальносистемним обмеженням на загальну кількість паралельних робітників. По суті, `max_parallel_workers_per_gather` контролює паралелізм для окремих запитів, тоді як `max_parallel_workers` і `max_worker_processes` задають загальні системні обмеження на паралельну обробку.[3]

Збільшення `max_parallel_workers_per_gather` дозволяє PostgreSQL використовувати більше робочих процесів для одного паралельного запиту, що потенційно прискорює виконання запиту. Однак, це також збільшує споживання ресурсів, тому його слід налаштовувати з обережністю.

Проведемо тестування з такими конфігураціями:

- max_parallel_workers_per_gather = 0
- max_parallel_workers_per_gather = 2 (за замовчування)
- max_parallel_workers_per_gather = 4

Тестований запит:

- SELECT COUNT(*) FROM rooms

Максимальна кількість рядків в таблиці: 1 000 000.

На графіку (Рисунок 5) видно, що використання більшої кількості паралельних робітників для запиту `SELECT COUNT(*) FROM rooms` виявилось більш затратним за часом, ніж використання меншої кількості робітників. Це пов’язано зі складністю виконання паралельних обчислень.

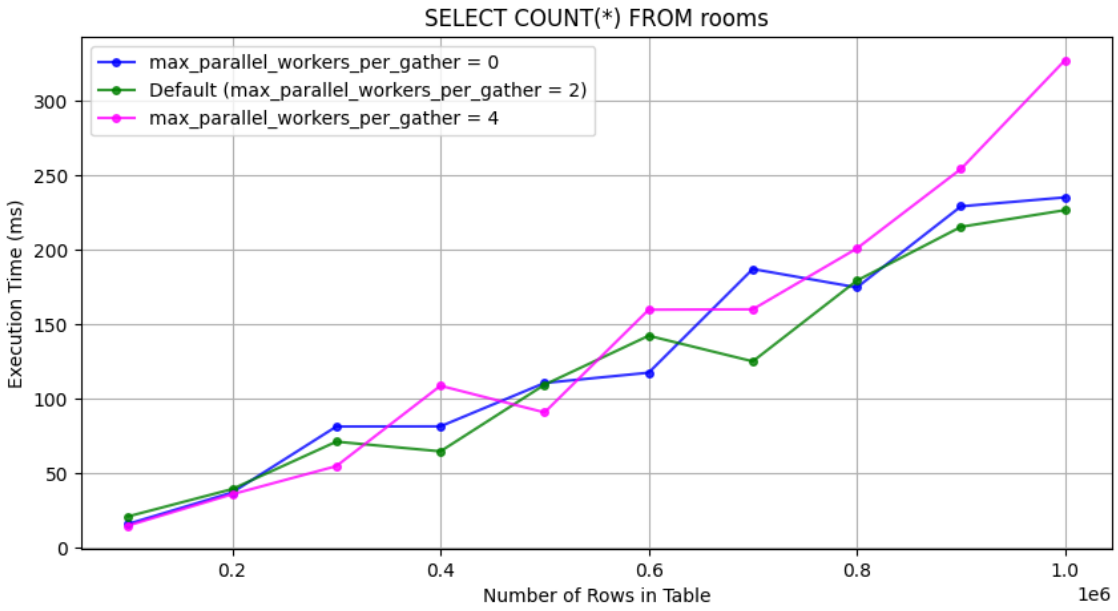

Рисунок 5

Кожен паралельний працівник повинен бути ініціалізований і йому повинні бути виділені ресурси, що включає в себе створення робочих процесів і керування ними. Цей процес ініціалізації додає накладних витрат, які можуть бути незначними для більш складних запитів, але можуть бути значними для простих запитів.

Головний процес повинен координувати роботу з декількома підлеглими процесами. Ця координація включає управління розподілом роботи, об'єднання результатів і забезпечення синхронізації, що може збільшити загальний час виконання.

Для паралельного запиту дані розбиваються на частини і розподіляються між працівниками. Для простого агрегату, такого як `COUNT(*)`, таке розбиття і розподіл даних може бути неефективним, оскільки накладні витрати на керування фрагментами можуть переважати над вигодами.

Після того, як кожен працівник обробить свій фрагмент, часткові результати мають бути об'єднані. У випадку підрахунку рядків, кожен працівник повертає результат, а керівник повинен підсумувати ці результати. Цей додатковий крок об'єднання часткових результатів збільшує час обробки.

Прості запити, такі як `SELECT COUNT(*)` мають низьку складність і не передбачають складних обчислень або великих перетворень даних. Підрахунок рядків у таблиці є простою задачею, яка не отримує значної вигоди від паралельної обробки, оскільки тут практично немає складних обчислень, які можна ефективно розпаралелити. Накладні витрати на паралелізм не виправдані, оскільки запит може бути виконаний дуже ефективно в однопотоковому режимі.

У нашому випадку:

- Використання 1 працівника (без паралелізму): Запит послідовно зчитує всі рядки і підраховує їх, що займає 200 мілісекунди.
- Використання 4 працівників: Кожен працівник читає приблизно 250 000 рядків. Однак, додаткові накладні витрати на ініціалізацію робітників, розподіл даних та об'єднання результатів можуть займати додатковий час. В результаті маємо загальний час 300 мілісекунд.

Спробуємо провести тестування на більш складному запиті:

- SELECT COUNT(*) FROM rooms GROUP BY type

На графіку (Рисунок 6) в цьому випадку найвиграшнішими виявились налаштування за замовчуванням, тобто використання 2 процесів. При значеннях 0 та 4 процеси результати майже однакові.

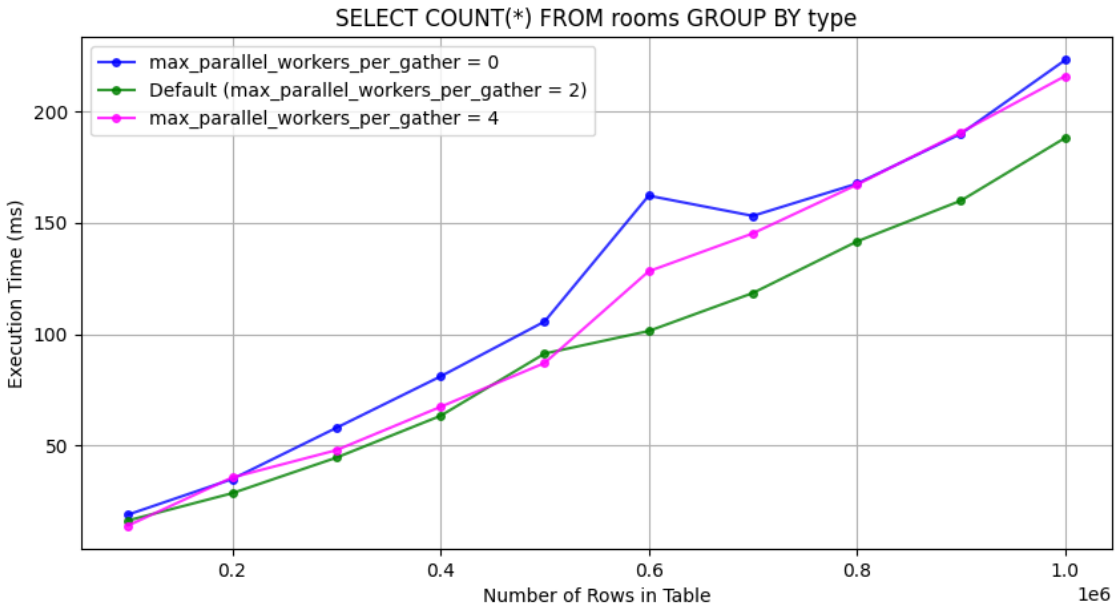

Рисунок 6

Проведемо тестування на ще більш складному запиті:

- SELECT h.id, h.name, r.price, COUNT(*) AS count
  FROM rooms r JOIN hotels h ON r.hotel_id = h.id
  WHERE r.price BETWEEN 200 AND 500 GROUP BY h.id, r.price

На графіку (Рисунок 7) на великих кількостях рядків непаралельне виконання виявилось не виграшним. Натомість значення 2 та 4 процеси показали кращі результати за попередні.

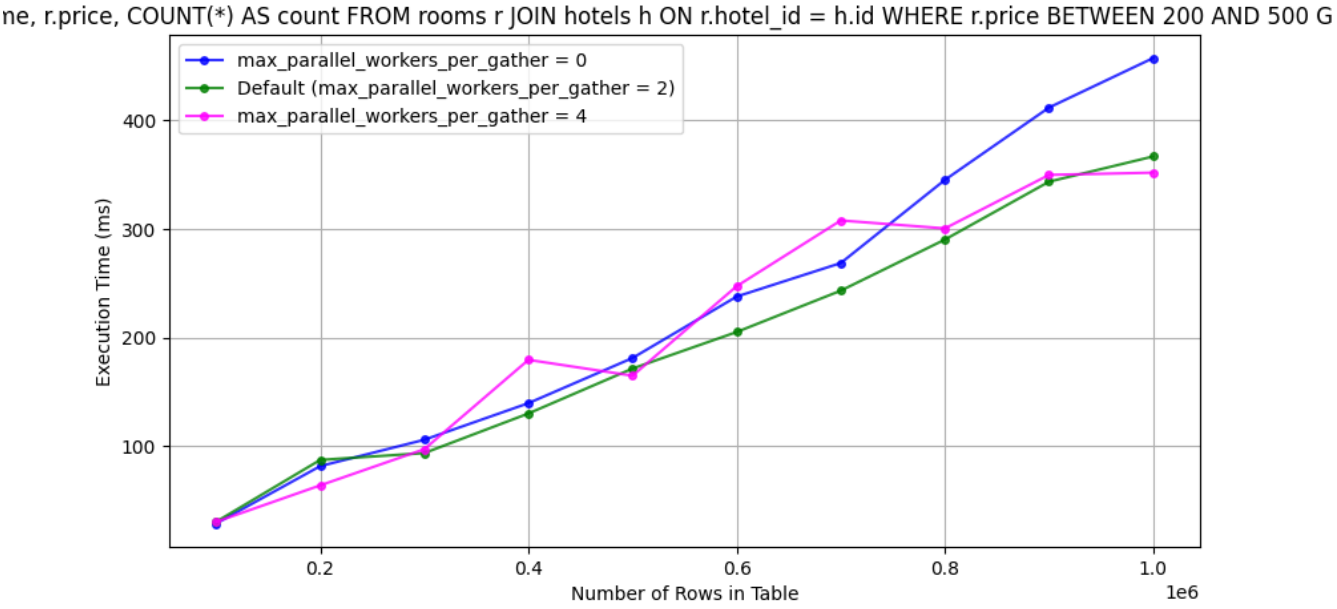

Рисунок 7

Підсумовуючи всі результати виконання різних запитів, можна зробити висновок, що значення за замовчуванням для параметру max_parallel_workers_per_gather виявилось найкращим для швидкого виконання запитів.

2.4.Висновки

У даному розділі було проведено детальний аналіз ключових параметрів конфігурації сервера PostgreSQL, які мають значний вплив на його продуктивність. Серед таких параметрів були розглянуті work_mem, shared_buffers та max_parallel_workers_per_gather.

Параметр work_mem визначає обсяг пам'яті, який виділяється для виконання операцій сортування та хешування в рамках одного запиту. Збільшення значення work_mem суттєво прискорює виконання запитів, які вимагають великих обсягів тимчасових даних. Разом з тим, надмірне збільшення цього параметра може призвести до перевищення доступної фізичної пам'яті, що негативно впливає на загальну продуктивність системи. Таким чином, оптимальне значення work_mem

повинно бути встановлено на основі конкретних потреб запитів та доступних ресурсів сервера.

Параметр shared_buffers визначає обсяг пам'яті, який PostgreSQL використовує для кешування даних. Це один з найбільш важливих параметрів, оскільки ефективне кешування дозволяє значно зменшити кількість дискових операцій і, відповідно, підвищити продуктивність. Рекомендовано встановлювати shared_buffers у межах 25-40% від загальної фізичної пам'яті сервера. Надто мале значення може призвести до частих звернень до диску, в той час як надто велике значення може залишити недостатньо пам'яті для інших процесів системи.

Параметр max_parallel_workers_per_gather визначає максимальну кількість процесів, які можуть бути використані для паралельного виконання одного запиту. Збільшення `max_parallel_workers_per_gather` не набагато покращує продуктивність паралельних запитів, але також призводить до збільшення використання ресурсів. Для простих запитів використання великої кількості робітників не є ефективним, оскільки є накладні витрати на налаштування та координацію, боротьба за ресурси. Для простих агрегованих операцій однопотокове виконання часто є більш ефективним завдяки меншим накладним витратам і простоті обробки.

3. Оптимізація запитів

Ефективність роботи з базами даних значною мірою залежить від оптимізації запитів. Вона передбачає виявлення та усунення слабких місць у процесі виконання запитів, щоб мінімізувати час, необхідний для отримання або оновлення даних.

Неоптимізовані запити можуть призвести до значного зниження продуктивності, збільшення часу очікування та неефективного використання ресурсів.

Передусім для оптимізації запитів важливо розуміти життєвий цикл запиту. Виконання запиту поділяється на такі етапи:

- Передача рядка запиту до бекенду бази даних
- Синтаксичний аналіз рядка запиту
- Планування запиту для оптимізації отримання даних
- Отримання даних з обладнання
- Передача результатів клієнту

[8]

1.1. Явні та неявні JOIN

Явний синтаксис JOIN дає деякий контроль над планувальником запитів. Щоб зрозуміти, чому це важливо, спочатку необхідно дати невелике пояснення.[9]

У простому запиті JOIN, наприклад:

- SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;

планувальник може об'єднати вказані таблиці в будь-якому порядку. Наприклад, він може згенерувати план запиту, який приєднує A до B за допомогою умови WHERE a.id = b.id, а потім приєднує C до своєї таблиці приєднання за допомогою іншої умови WHERE. Крім того, він може об'єднати B і C, а потім приєднати A до результату.

Однак це неефективно, оскільки вимагає формування повного декартового добутку A і C, а в умові WHERE немає відповідної умови для оптимізації об'єднання.

Важливим моментом є те, що ці варінти об'єднання дають семантично еквівалентні результати, але витрати на виконання можуть бути дуже різними. Тому планувальник враховує всі ці можливості, щоб знайти найефективніший план запиту.

Якщо запит охоплює лише дві або три таблиці, то не потрібно турбуватися про велику кількість можливих об'єднаннь. Однак, їхня кількість зростає в геометричній прогресії зі збільшенням кількості таблиць. Коли вхідних таблиць більше 10-ти, вичерпний перебір усіх можливих варіантів стає недоцільним.

Якщо вхідних таблиць занадто багато, планувальник PostgreSQL перемикається з вичерпного пошуку на генетичний імовірнісний пошук через обмежену кількість можливостей (поріг перемикання задається параметром часу виконання geqo_threshold).

Якщо запит містить зовнішні об'єднання, планувальник має менше ступенів свободи, ніж для простих (внутрішніх) об'єднань. Розглянемо наступний приклад:

- SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

Обмеження в цьому запиті схожі на попередній приклад, але семантика відрізняється. Порядок об'єднання не такий, як у попередньому прикладі. Тому планувальник не може вибрати порядок об'єднання. Тому цей запит потребує менше часу на планування, ніж попередній. В інших випадках планувальник може вирішити, що один або декілька порядків об'єднання є кращими. Наприклад:

- SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);

у цьому випадку A може бути приєднаний до B або C першим. Наразі лише FULL JOIN повністю обмежує порядок з'єднань.

Внутрішні JOIN (JOIN, INNER JOIN, CROSS JOIN), записані явно, не впливають на порядок з'єднання таблиць. Це пов'язано з тим, що вони еквівалентні простому перерахуванню таблиць у FROM. Однак можна вказати планувальнику розглядати всі JOIN як такі, що обмежують порядок з'єднання. Для цього параметр join_collapse_limit можна встановити рівним 1. Розглянемо наступний приклад:

- SELECT * FROM a CROSS JOIN b, c, d, e WHERE ... ;

Якщо join_collapse_limit = 1, планувальник приєднає A до B, а потім до іншої таблиці. У цьому прикладі кількість можливих замовлень на приєднання зменшується в 5 разів.

Обмеження пошуку планувальника у такий спосіб є корисною технікою для скорочення часу планування і спрямування планувальника до правильного плану запитів. Якщо планувальник за замовчуванням вибирає поганий порядок об'єднання, синтаксис JOIN можна використати для того, щоб змусити його вибрати кращий порядок об'єднання.

Неявний JOIN на прикладі тестової бази даних:

- Перше виконання:

```
postgres=# EXPLAIN ANALYZE
                SELECT *
                FROM rooms r, bookings b, users u, profiles p
                WHERE u.id = b.user_id
                AND r.id = b.room_id
                AND p.user_id = u.id;
                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Hash Join  (cost=245.42..511.01 rows=7902 width=430) (actual time=3.524..13.405 rows=7000 loops=1)
   Hash Cond: (bookings.user_id = users.id)
```

-> Hash Join (cost=166.99..415.76 rows=6399 width=339) (actual time=2.246..9.432 rows=7000 loops=1)

Hash Cond: (bookings.room_id = rooms.id)

-> Hash Join (cost=67.48..299.42 rows=6399 width=158) (actual time=1.002..5.540 rows=7000 loops=1)

Hash Cond: (bookings.user_id = profiles.user_id)

-> Seq Scan on bookings (cost=0.00..212.80 rows=7280 width=129) (actual time=0.018..1.995 rows=7000 loops=1)

-> Hash (cost=38.88..38.88 rows=2288 width=29) (actual time=0.916..0.918 rows=2000 loops=1)

Buckets: 4096 Batches: 1 Memory Usage: 155kB

-> Seq Scan on profiles (cost=0.00..38.88 rows=2288 width=29) (actual time=0.033..0.447 rows=2000 loops=1)

-> Hash (cost=73.67..73.67 rows=2067 width=181) (actual time=1.152..1.153 rows=1000 loops=1)

Buckets: 4096 Batches: 1 Memory Usage: 246kB

-> Seq Scan on rooms (cost=0.00..73.67 rows=2067 width=181) (actual time=0.114..0.727 rows=1000 loops=1)

-> Hash (cost=52.08..52.08 rows=2108 width=91) (actual time=1.206..1.207 rows=2000 loops=1)

Buckets: 4096 Batches: 1 Memory Usage: 285kB

-> Seq Scan on users (cost=0.00..52.08 rows=2108 width=91) (actual time=0.015..0.584 rows=2000 loops=1)

Planning Time: 3.624 ms

Execution Time: 14.029 ms

(18 rows)

Спочатку планувальник об’єднує profiles з bookings, потім (profiles, books) з rooms, (profiles, bookings, rooms) з users. Тобто, маємо такий порядок:

profiles->bookings->rooms->users

- Друге виконання:

```
postgres=# EXPLAIN ANALYZE
            SELECT *
            FROM rooms r, bookings b, users u, profiles p
            WHERE u.id = b.user_id
            AND r.id = b.room_id
            AND p.user_id = u.id;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=212.50..477.79 rows=7000 width=430) (actual time=4.397..16.373 rows=7000 loops=1)
   Hash Cond: (users.id = profiles.user_id)
   -> Hash Join (cost=151.50..398.37 rows=7000 width=401) (actual time=3.222..12.591 rows=7000 loops=1)
         Hash Cond: (bookings.user_id = users.id)
         -> Hash Join (cost=75.50..303.95 rows=7000 width=310) (actual time=1.589..8.328 rows=7000 loops=1)
               Hash Cond: (bookings.room_id = rooms.id)
               -> Seq Scan on bookings  (cost=0.00..210.00 rows=7000 width=129) (actual time=0.017..2.600 rows=7000 loops=1)
               -> Hash (cost=63.00..63.00 rows=1000 width=181) (actual time=1.500..1.502 rows=1000 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 222kB
                     -> Seq Scan on rooms  (cost=0.00..63.00 rows=1000 width=181) (actual time=0.087..0.715 rows=1000 loops=1)
```

-> Hash (cost=51.00..51.00 rows=2000 width=91) (actual time=1.526..1.526 rows=2000 loops=1)

Buckets: 2048 Batches: 1 Memory Usage: 269kB

-> Seq Scan on users (cost=0.00..51.00 rows=2000 width=91) (actual time=0.019..0.546 rows=2000 loops=1)

-> Hash (cost=36.00..36.00 rows=2000 width=29) (actual time=1.042..1.043 rows=2000 loops=1)

Buckets: 2048 Batches: 1 Memory Usage: 139kB

-> Seq Scan on profiles (cost=0.00..36.00 rows=2000 width=29) (actual time=0.029..0.370 rows=2000 loops=1)

Planning Time: 4.157 ms

Execution Time: 17.035 ms

(18 rows)

Спочатку планувальник об’єднує rooms з bookings, потім (rooms, books) з users, (rooms, bookings, users) з profiles. Тобто, маємо такий порядок:

rooms->bookings->users->profiles

Застосуємо цей порядок для явного JOIN на прикладі тестової бази даних:

- postgres=# EXPLAIN ANALYZE

```
SELECT *
FROM (
    users u LEFT JOIN
    (bookings b LEFT JOIN rooms r on b.room_id = r.id)
    ON b.user_id = u.id
)
LEFT JOIN profiles p ON p.user_id = u.id;
```

QUERY PLAN

```
 Hash Left Join  (cost=212.50..477.79 rows=7000 width=430) (actual
time=1.752..14.401 rows=7000 loops=1)
   Hash Cond: (u.id = p.user_id)
   ->  Hash Left Join  (cost=151.50..398.37 rows=7000 width=401) (actual
time=0.919..10.487 rows=7000 loops=1)
         Hash Cond: (b.user_id = u.id)
         ->  Hash Left Join  (cost=75.50..303.95 rows=7000 width=310) (actual
time=0.397..6.079 rows=7000 loops=1)
               Hash Cond: (b.room_id = r.id)
               ->  Seq Scan on bookings b  (cost=0.00..210.00 rows=7000 width=129) (actual
time=0.010..1.335 rows=7000 loops=1)
               ->  Hash  (cost=63.00..63.00 rows=1000 width=181) (actual time=0.373..0.375
rows=1000 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 223kB
                     ->  Seq Scan on rooms r  (cost=0.00..63.00 rows=1000 width=181) (actual
time=0.019..0.189 rows=1000 loops=1)
         ->  Hash  (cost=51.00..51.00 rows=2000 width=91) (actual time=0.517..0.517
rows=2000 loops=1)
               Buckets: 2048  Batches: 1  Memory Usage: 269kB
               ->  Seq Scan on users u  (cost=0.00..51.00 rows=2000 width=91) (actual
time=0.004..0.208 rows=2000 loops=1)
   ->  Hash  (cost=36.00..36.00 rows=2000 width=29) (actual time=0.821..0.822
rows=2000 loops=1)
         Buckets: 2048  Batches: 1  Memory Usage: 139kB
         ->  Seq Scan on profiles p  (cost=0.00..36.00 rows=2000 width=29) (actual
time=0.003..0.337 rows=2000 loops=1)
 Planning Time: 0.714 ms
 Execution Time: 14.954 ms
```

(18 rows)

1.2. Індекси

Індекс у PostgreSQL - це структура бази даних, яка підвищує швидкість операцій пошуку даних у таблиці за рахунок додаткового місця для запису та зберігання даних індексу.

Індекси ефективно визначають місцезнаходження даних без необхідності сканувати кожен рядок таблиці. Вони корисні лише тоді, коли кількість рядків, які потрібно отримати з таблиці, відносно невелика.[10] Якщо у запиті очікується отримання великої кількості рядків, планувальник обере послідовне сканування, ніж сканування за допомогою індексу.

При назначені для колонки UNIQUE або PRIMARY KEYобмеження індекси для цих колонок створюються автоматично:

- postgres=# CREATE TABLE "rooms"(
  "id" uuid,
  "hotel_id" uuid,
  "number" int NOT NULL,
  PRIMARY KEY ("id"),
  FOREIGN KEY ("hotel_id") REFERENCES "hotels"("id"),
  UNIQUE ("number"));
- postgres=# SELECT indexname FROM pg_indexes WHERE tablename =
  'rooms';
  indexname
  ----
  rooms_pkey
  rooms_number_key
  (2 rows)

PostgreSQL підтримує такі типів індексів, які призначені для підвищення продуктивності різних запитів: B-Tree, Hash, GiST, SP-GiST, GIN, BRIN.[4]

B-Tree є типом за замовчуванням, коли індекс створюється за допомогою команди CREATE INDEX… без вказання типу. Він організований у вигляді деревоподібної структури. Індекс починається з кореневого вузла з вказівниками на дочірні вузли. Кожен вузол дерева зазвичай містить декілька пар ключ-значення, де ключі використовуються для індексування, а значення вказують на відповідні дані в таблиці. Використовується коли індексований стовпець бере участь у порівнянні за допомогою одного з цих операторів:

- <
- <=
- =
- >=
- >
- BETWEEN
- IN
- IS NULL
- IS NOT NULL

Hash зберігає 32-бітний хеш-код, отриманий зі значення індексованого стовпця. Обслуговування хеш-індексів може бути дорожчим, ніж індексів B-Tree, через необхідність вирішувати колізії та перехешувати дані.[11] Використовується коли індексований стовпець бере участь у порівнянні за допомогою оператора рівності: « = ».

GiST (Generalized Search Tree) - узагальнене дерево пошуку, корисне для багатовимірних даних. Цей тип використовуються для просторових даних, зіставлення триграм і повнотекстового пошуку.

GIN (Generalized Inverted Index) - узагальнений інвертований індекс. Цей тип індексу призначений для повнотекстового пошуку та інших складних сценаріїв пошуку, таких як масиви та типи даних JSONB.

SP-GiST (Space-partitioned Generalized Search Tree) - просторово-розділене узагальнене дерево пошуку. Цей тип індексу розширює тип GiST, пропонуючи ефективне індексування для спеціалізованих типів даних або структур даних.

BRIN (Block Range INdexes) - індекси діапазону блоків. Цей тип індексу призначений для дуже великих таблиць, оскільки забезпечують компактне та ефективне індексування на основі діапазонів блоків. BRIN є найбільш ефективним для стовпців, значення яких добре корелюють з фізичним порядком рядків таблиці.[10]

Протестуємо використання індексів типу B-Tree та типу Hash на запиті:

- SELECT * FROM rooms WHERE hotel_id=<hotel-id>;

Де <hotel-id> - ідентифікатор будь-якого готелю в таблиці hotels

Для того, щоб спростити план виконання запиту, встановимо нульове значення для параметру max_parallel_workers_per_gather, який контролює максимальну кількість робочих процесів, які можуть бути запущені одним вузлом Gather або Gather Merge у плані запиту:

- SET max_parallel_workers_per_gather = 0;

Протестуємо виконання запиту спочатку без використання індексу, а потім з індексами B-Tree та Hash типів:

- CREATE INDEX "rooms_hotel_id_ix" ON rooms USING BTREE(hotel_id);
- CREATE INDEX "rooms_hotel_id_ix" ON rooms USING HASH(hotel_id);

Результати тестування показали, що використання індексу значно пришвидшує виконання SELECT запиту (Рисунок 8).

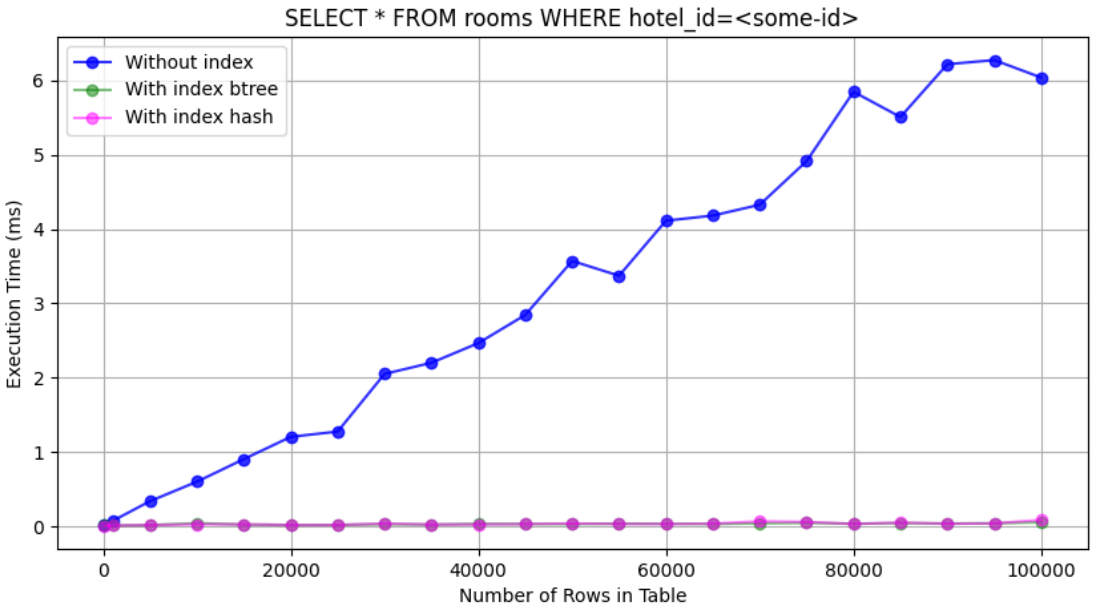

Рисунок 8

Різниця в продуктивності запитів між B-Tree та Hash індексами для стовпців UUID настільки мала, що її майже не помітно порівняно із запитами без індексів. Але якщо збільшити їхні графіки, то можна помітити, що в більшості випадків Hash індекс має перевагу (Рисунок 9).

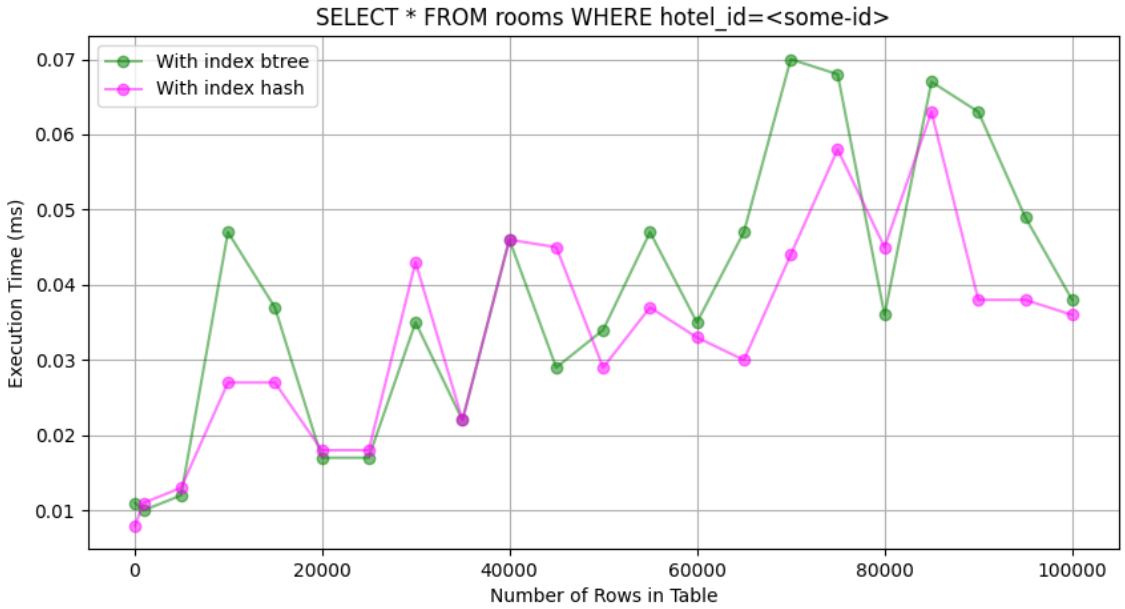

Рисунок 9

Хоч індексування і прискорює отримання даних, водночас воно уповільнює створення, оновлення та видалення рядків, оскільки індекс також потребує оновлення.[10]

Проведемо тестування для INSERT запиту в rooms таблиці:

- INSERT INTO rooms(id, hotel_id, name, number, price, floor, beds_number, type, is_smoking_allowed) VALUES (…);

Як і раніше, ми тут використали один додадковий індекс для hotel_id, тож різниця не є дуже помітною (Рисунок 10).

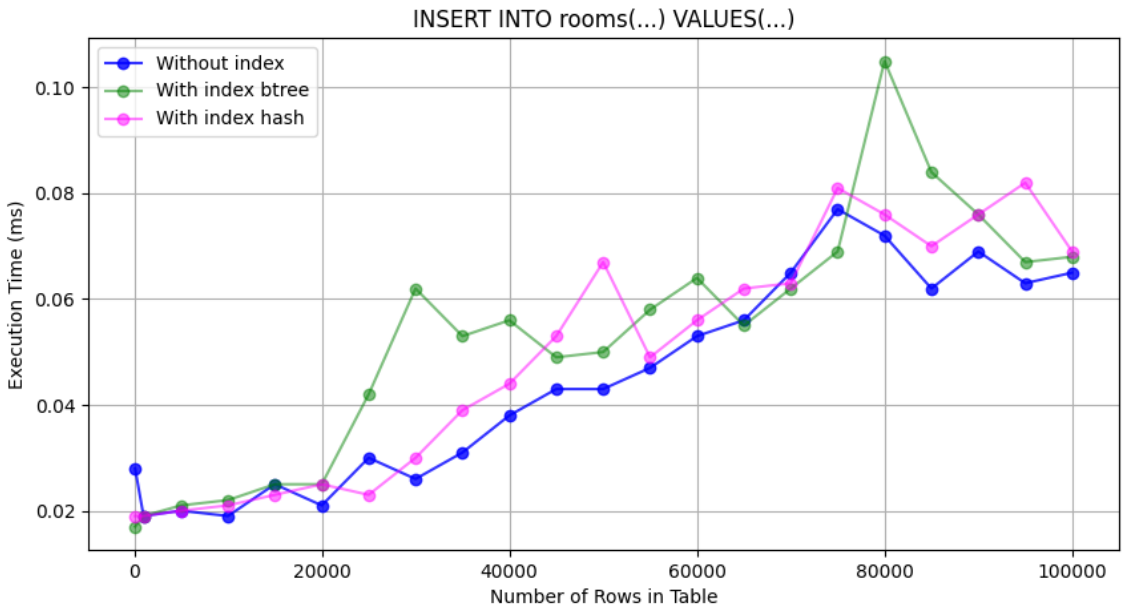

Рисунок 10 Виконання запиту INSERT з 1-им додатковим індексом

Уповільнення операцій запису буде помітно при великій кількості індексів відносно однієї таблиці. Створимо додаткові B-Tree індекси для 4 стовбців:

- CREATE INDEX "rooms_hotel_id_ix" ON rooms USING BTREE(hotel_id);
- CREATE INDEX "rooms_name_ix" ON rooms USING BTREE(name);
- CREATE INDEX "rooms_beds_number_ix" ON rooms USING BTREE(beds_number);
- CREATE INDEX "rooms_type_ix" ON rooms USING BTREE(type);

А потім Hash індекси для цих же стовбців:

- CREATE INDEX "rooms_hotel_id_ix" ON rooms USING HASH(hotel_id);
- CREATE INDEX "rooms_name_ix" ON rooms USING HASH (name);
- CREATE INDEX "rooms_beds_number_ix" ON rooms USING HASH (beds_number);
- CREATE INDEX "rooms_type_ix" ON rooms USING HASH (type);

Проведемо тестування для INSERT запиту в rooms таблиці, де задіяні всі додаткові індексовані стовбці:

- INSERT INTO rooms(id, **hotel_id, name**, number, price, floor, **beds_number, type**, is_smoking_allowed) VALUES (…);

За результатами тестування бачимо, що B-Tree індекси не сильно вплинули на INSERT операцію порівняно з HASH індексами (Рисунок 11).

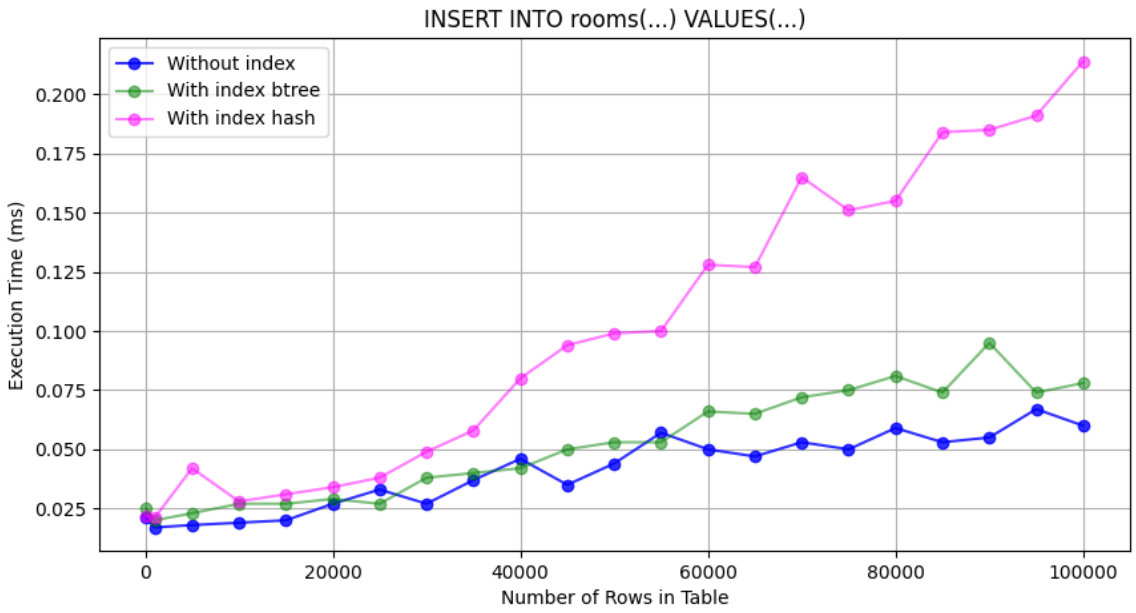

Рисунок 11 Виконання запиту INSERT з 4-ма додатковими індексами

1.2.1. Index only scans

У PostgreSQL індекси не є частиною основних даних таблиці. Їх зберігають окремо, називаючи основну таблицю «купою» (heap). При звичайному скануванні за індексом, для кожного рядка потрібно витягувати дані з індексу, а потім з купи.[12]

Хоча записи індексу, що відповідають умові WHERE, розташовані близько один до одного, рядки таблиці, на які вони посилаються, можуть бути розкидані по всій купі. Через це, доступ до купи під час сканування індексу передбачає багато випадкових звернень, що може бути повільним, особливо на HDD.

Для вирішення цієї проблеми PostgreSQL підтримує «сканування лише за індексом» (Index only scans). Це означає, що запити можуть бути виконані лише за даними індексу, без доступу до купи. Замість того, щоб звертатися до рядків у купі, значення отримуються безпосередньо з кожного запису індексу.

Сканування лише за індексом значно швидше, ніж звичайне сканування, адже не потрібен доступ до купи. Завдяки ньому зменшується навантаження на диск, що покращує загальну продуктивність.

Приклад Index Scan:

- postgres=# EXPLAIN ANALYZE SELECT * FROM rooms WHERE id = 'a9c17097-6581-51b8-9372-6d1e708780ca';

QUERY PLAN

Index Scan using rooms_pkey on rooms (cost=0.42..8.44 rows=1 width=181) (actual time=0.126..0.128 rows=1 loops=1)

Index Cond: (id = 'a9c17097-6581-51b8-9372-6d1e708780ca'::uuid)

Planning Time: 0.517 ms

Execution Time: 0.314 ms

(4 rows)

Приклад Index Only Scan:

- postgres=# EXPLAIN ANALYZE SELECT id FROM rooms WHERE id = 'a9c17097-6581-51b8-9372-6d1e708780ca';

QUERY PLAN

Index Only Scan using rooms_pkey on rooms (cost=0.42..4.44 rows=1 width=16) (actual time=0.106..0.108 rows=1 loops=1)

Index Cond: (id = 'a9c17097-6581-51b8-9372-6d1e708780ca'::uuid)

Heap Fetches: 0

Planning Time: 0.100 ms

Execution Time: 0.135 ms

(5 rows)

Проведемо тестування данного запиту з максимальною кількість рядків в таблиці 1 000 000.

З графіку (Рисунок 12) видно, що сканування лише за індексом набагато швидше, ніж звичайне сканування (сканування таблиці індекса та основної таблиці).

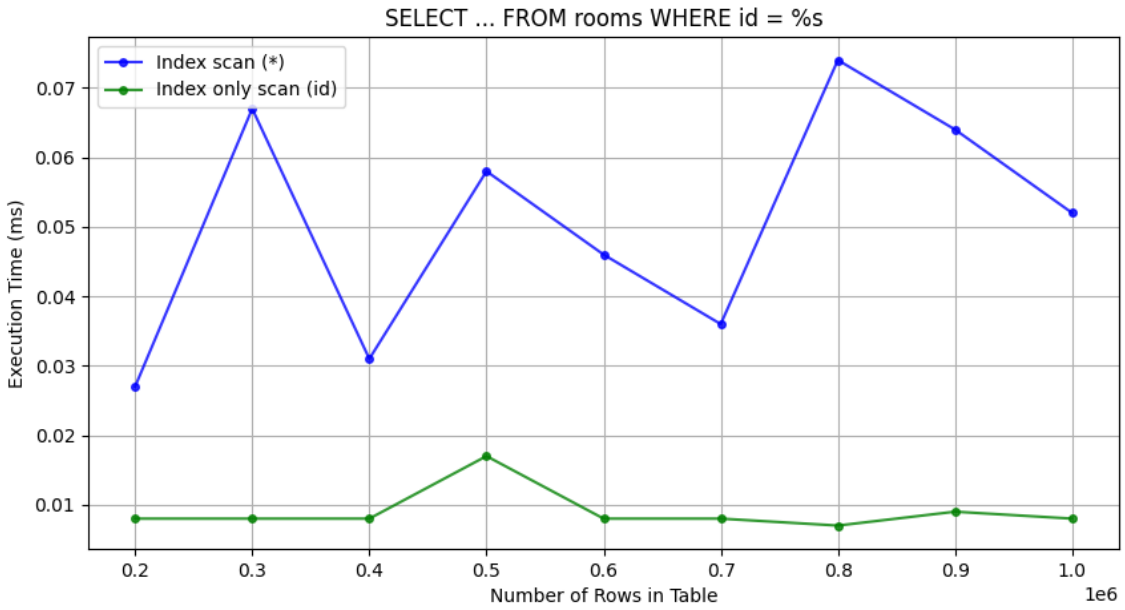

Рисунок 12

Однак таке сканування можливе не для всіх запитів. Для роботи цієї функції індекс повинен містити всі необхідні стовпці для відповіді на запит.

3.1.Висновки

У даному розділі було досліджено різні аспекти оптимізації запитів у PostgreSQL, зокрема, явні та неявні JOIN, використання індексів та технологію Index-Only Scans.

Явні JOIN (explicit JOIN) вказуються безпосередньо в запиті за допомогою ключових слів INNER JOIN, LEFT JOIN, RIGHT JOIN тощо. Вони дозволяють розробникам точно визначити логіку об’єднання таблиць і можуть покращити читабельність коду.

Неявні JOIN (implicit JOIN) використовують синтаксис, де об’єднання таблиць відбувається за допомогою перерахування таблиць у секції FROM і вказання умов об’єднання в секції WHERE. Хоча такий підхід є менш виразним, він може бути зручним для простих запитів.

Використання явних JOIN пришвидшує виконання запиту, оскільки планувальнику не треба визначати шляхом перебору який порядок об’єднання є найкращим для даного запиту.

Індекси суттєво покращують продуктивність запитів, прискорюючи операції пошуку та сортування. Вони особливо корисні для колонок, які часто використовуються в умовах WHERE, JOIN та ORDER BY.

Створення індексів повинно бути обґрунтоване аналізом частоти використання конкретних колонок у запитах. Надмірна кількість індексів може призвести до збільшення часу вставки та оновлення даних, а також до збільшення використання пам’яті.

Index-Only Scans є потужним механізмом оптимізації, який дозволяє виконувати запити, використовуючи лише індекс, без звернення до основної таблиці. Це можливо, коли всі необхідні дані для запиту містяться в індексі.

Використання Index-Only Scans значно зменшує кількість дискових операцій, що суттєво підвищує продуктивність запитів.

У підсумку, оптимізація запитів за допомогою правильної комбінації JOIN, індексів та Index-Only Scans є критично важливою для забезпечення високої продуктивності СУБД PostgreSQL. Ефективне використання явних та неявних JOIN дозволяє налаштувати оптимальний шлях об'єднання таблиць, а правильно налаштовані індекси та Index-Only Scans значно прискорюють доступ до даних. Такий підхід забезпечує швидке виконання запитів і знижує навантаження на сервер, що є важливим для підтримки стабільної та ефективної роботи бази даних.

СПИСОК ВИКОРСТАНИХ ДЖЕРЕЛ

1. Docker Compose overview. *Docker Documentation*. URL: https://docs.docker.com/compose/.
2. Seed. *Snaplet*. URL: https://www.snaplet.dev/seed.
3. 20.4. Resource Consumption. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/runtime-config-resource.html.
4. Tuning Your PostgreSQL Server - PostgreSQL wiki. *PostgreSQL wiki*. URL: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server.
5. F.27. pg_buffercache – inspect PostgreSQL buffer cache state. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/pgbuffercache.html.
6. Optimizing PostgreSQL shared_buffers - PostgreSQL High Performance Guide (Part 2/12) - Distributed Systems Authority. *Distributed Systems Authority*. URL: https://distributedsystemsauthority.com/optimizing-postgresql-shared-buffers/.
7. F.27. pg_buffercache – inspect PostgreSQL buffer cache state. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/pgbuffercache.html.
8. Wiles F. Performance Tuning PostgreSQL. *REVSYS*. URL: https://www.revsys.com/writings/postgresql-performance.html.
9. 14.3. Controlling the Planner with Explicit JOIN Clauses. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/explicit-joins.html.
10. Efficient Use of PostgreSQL Indexes | Heroku Dev Center. *Heroku Dev Center*. URL: https://devcenter.heroku.com/articles/postgresql-indexes.
11. Oyama F. Advanced Indexing Strategies in PostgreSQL. *freeCodeCamp.org*. URL: https://www.freecodecamp.org/news/postgresql-indexing-strategies/.
12. 11.9. Index-Only Scans and Covering Indexes. *PostgreSQL Documentation*. URL: https://www.postgresql.org/docs/current/indexes-index-only-scans.html.