Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра "Системний аналіз та обчислювальна математика"
(повне найменування кафедри)

Пояснювальна записка

до дипломної роботи
магістра
(ступінь вищої освіти)

на тему Оптимізація запитів до баз даних з різними структурами та обсягом даних
(назва теми)

Виконав: студент 2 курсу, групи 813м

Спеціальності 124 - Системний аналіз
(код і найменування спеціальності)

Освітня програма (спеціалізація)
"Інтелектуальні технології та прийняття рішень в складних системах"

МЕХРЯКОВ Є. В.
(ПРІЗВИЩЕ та ініціали)

Керівник БАХРУШИН В. Є.
(ПРІЗВИЩЕ та ініціали)
Рецензент ЛОЗОВСЬКА Л.І.
(ПРІЗВИЩЕ та ініціали)

2024

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Факультет комп'ютерних наук і технологій
Кафедра "Системний аналіз та обчислювальна математика"
Ступінь вищої освіти бакалавр
Спеціальність 124 - Системний аналіз
(код і найменування)
Освітня програма (спеціалізація) Інтелектуальні технології та прийняття рішень в складних системах
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри Терещенко Е.В.
«20» січня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА

МЕХРЯКОВА Євгенія Володимировича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема роботи Оптимізація запитів до баз даних з різними структурою та обсягом даних
керівник роботи БАХРУШИН Володимир Євгенійович
(прізвище, ім'я по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від «20» листопада 2024 року №480
2. Строк подання студентом роботи 20 грудня 2024
3. Вихідні дані до роботи мова запитів SQL, СУБД Microsoft SQL Server 2022
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) аналіз існуючих джерел інформації за можливими напрямками, які пов'язані з темою дипломної роботи, виокремлення найбільш ефективних методів оптимізації, визначення їх ефективності у різних ситуаціях та при різних умовах, порівняння результатів виконання різноманітних запитів до і після застосування представлених методів оптимізації
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

6. Консультанти розділів роботи

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
Розділ 1, Розділ 2	БАХРУШИН В.Є., проф. каф. САОМ	23 серпня 2024	19 січня 2025
Нормконтроль	ШИРОКОРАД Д.В., ст. викладач	20 січня 2025	20 січня 2025

7. Дата видачі завдання «23» серпня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Визначення основних пунктів та вмісту до них	23.08.2024	
2	Дослідження сучасних методів написання запитів	10.09.2024	
3	Виконати пошук та визначити недоліки аналогів	21.09.2024	
4	Визначитись зі способом організації роботи	12.10.2024	
5	Надати приклади виконання роботи та результатів	15.11.2024	
6	Проведення експериментів для практичної частини	11.11.2024	
7	Написати висновок	23.12.2024	

Студент Підпис Євгеній МЕХРЯКОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник роботи Підпис Володимир БАХРУШИН
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Дипломна робота: 52 с., 23 рис., 11 джерел.

Об’єкт дослідження – запити до баз даних з різними структурами та обсягами даних.

Предмет дослідження – методи оптимізації запитів до різних типів баз даних.

Мета роботи – дослідження прийомів і стратегій оптимізації запитів до баз даних для підвищення їх продуктивності та ефективності.

Методи дослідження – поглиблений аналіз планів виконання запитів, оцінка витрат ресурсів та апробація різних методів у тестовому середовищі Microsoft SQL Server з використанням SQL Server Management Studio.

Актуальність теми полягає у зростанні обсягів даних та різноманітті їх структур. Сучасні бази даних потребують ефективних запитів для забезпечення швидкого доступу та оптимізації ресурсів.

Практичне значення роботи виявляється у розробці рекомендацій для оптимізації запитів сучасних баз даних для розробників та адміністраторів. Впровадження отриманих результатів сприятиме підвищенню продуктивності та стабільності систем управління даними.

ЗАПИТ, БАЗА ДАНИХ, ОПЕРАТОРИ, СТРУКТУРА, ДАНІ, ЕФЕКТИВНІСТЬ, SQL, MICROSOFT SQL SERVER

ЗМІСТ

ВСТУП

З наступом цифрової епохи, почалося активне використання великих обсягів інформації і невдовзі були створені перші системи управління баз даних, які виявились найкращим способом для зберігання будь-яких видів даних, що відкрило величезну кількість нових можливостей для всіх сфер діяльності людства.

Сьогодні, весь процес від побудови бази даних до її використання через запити є дуже складним, маючи багато важливих нюансів та правил, які обов’язково необхідно дотримуватись для ефективності роботи, але на жаль існує замало джерел інформації, які б могли надати структурований список методів, що допомагали б в цих задачах.

Мета цієї роботи полягає у створенні списку рекомендацій, які варто притримуватись під час роботи з базами даних і при написанні будь-яких запитів до них, з метою значного пришвидшення часу виконання, оптимального використання простору на серверах та полегшення роботи при розширеннях систем у майбутньому.

З завдань, які необхідно було вирішити під час роботи, можна виділити аналіз існуючих джерел інформації за можливими напрямками, які пов’язані з темою дипломної роботи, виокремлення найбільш ефективних методів оптимізації, визначення їх ефективності у різних ситуаціях та при різних умовах, порівняння результатів виконання різноманітних запитів до і після застосування представлених методів оптимізації.

1 ТЕОРЕТИЧНА ЧАСТИНА

1.1 Структури баз даних та мови створення запитів

Бази даних є основою сучасних систем керування даними, головними факторами, які їх розділяють є структури їх реалізації та мова запитів, яка використовується.

Базова структура бази даних визначає, як дані індексуються, зберігаються та доступні, впливаючи на продуктивність запитів. Загалом, структури баз даних можна розділити на два основні види: реляційні (SQL) та нереляційні (NoSQL) [1], також можна виділити кілька підвидів, як графові та ієрархічні. Реляційні бази даних вимагають стратегій оптимізації, таких як індексування та нормалізація, щоб підвищити ефективність. Бази даних NoSQL навпаки, виграють від моделювання даних, адаптованого до конкретних шаблонів запитів, наголошуючи на денормалізації та розділенні. Ієрархічні бази даних покладаються на методи доступу на основі шляхів, часто вимагаючи попередньо обчислених шляхів для швидшого обходу. Кожна з цих структур розроблена відповідно до певних вимог до зберігання та доступу до даних, тому для оптимізованої роботи запитів, важливо зрозуміти їх сильні сторони та відмінності. Зважаючи на мету цієї дипломної роботи, буде оптимально розглядати саме реляційні бази даних, через більшу популярність та функціонал.

Реляційні бази даних організовують дані в заздалегідь визначені таблиці, що складаються з рядків і стовпців із зв’язками, встановленими за допомогою ключів. Ця структура є ефективною для забезпечення цілісності та узгодженості даних. Широко використовувані системи управління реляційними базами даних (RDBMS) включають MySQL, PostgreSQL і Microsoft SQL Server, кожен з яких підтримує SQL (структуровану мову запитів) як стандарт для запитів і керування даними. Реляційні бази даних особливо підходять для структурованих даних і транзакційних додатків, але їх продуктивність може погіршитися через

неоптимізовані запити, особливо у випадку складних об’єднань або великих наборів даних [2].

Відразу після виду баз даних, найважливішим фактором є мови запитів, які є основними інструментами для взаємодії з БД, надаючи засоби для отримання, обробки та аналізу даних. Кожен тип бази даних використовує певну мову запитів, розроблену відповідно до її архітектури та випадків використання. Наприклад, реляційні бази даних в основному покладаються на SQL (Structured Query Language) [3], який забезпечує зрозумілий синтаксис для визначення та обробки даних за допомогою таких операторів, як *SELECT*, *INSERT*, *UPDATE* і *DELETE*. Структурований характер і стандартизація SQL роблять його дуже ефективним для операцій, що включають складні зв’язки між таблицями.

Протилежно до реляційних, нереляційні (NoSQL) бази даних, які включають бази даних документів, ключ-значення, стовпчасті та графові бази даних, часто використовують спеціалізовані механізми запитів, адаптовані до їхніх конкретних моделей даних. Бази даних, орієнтовані на документи, такі як MongoDB, використовують синтаксис запитів на основі JSON-подібних структур, пропонуючи гнучкість для неструктурованих або напівструктурованих даних. Сховища ключ-значення, такі як Redis, підтримують мінімалістичні запити, зосереджені на отриманні значень, пов’язаних із ключами, віддаючи пріоритет простоті та продуктивності.

Графові бази даних, такі як Neo4j, використовують такі мови, як Cypher, призначені для вираження шаблонів у структурах графів. Ці мови застосовуються в сценаріях, де стосунки та зв’язки між об’єктами є центральними, наприклад соціальні мережі чи системи рекомендацій. Бази даних із сімейством стовпців, такі як Cassandra, використовують CQL (Cassandra Query Language), який нагадує SQL, але оптимізований для розподіленої архітектури та масштабованості систем NoSQL [4].

Відмінності в мовах запитів також впливають на стратегії оптимізації. Запити SQL часто покладаються на плани виконання, створені системами керування базами даних, які враховують такі фактори, як індекси та стратегії

об’єднання. У системах NoSQL оптимізація може включати сегментування, реплікацію або розробку схеми бази даних, щоб мінімізувати потребу у складних запитах.

1.2 Алгоритми роботи запитів та основи їх оптимізації

До початку оптимізації та пропонування дій для покращення роботи запитів, варто спочатку дослідити алгоритми роботи найчастіше використовуваних запитів, щоб зрозуміти їх структуру та принципи роботи. З усіх видів запитів, які використовують, можна окремо за частотою використання можна виділити оператори SELECT (оператор виборки/отримання даних з таблиць) та INSERT (внесення нових даних до існуючої таблиці), і навіть з цих двох, викборка буде в сотні, а то й в тисячі разів більш вживана, як простий приклад, можна привести сторінку онлайн-магазину, де оператор INSERT буде використовуватись при створенні нового користувача та додавання товару, коли SELECT буде спрацьовувати під час кожного перегляду каталогу відвідувачем сайту. І саме з цих причин є очевидний сенс розглядати саме SELECT операцію та її виконання.

Алгоритми виконання запиту виборки у більшості систем управління базами даних однаковий, або щонайменше схожий на наступний:

1. Синтаксичний аналіз запиту: СУБД аналізує SQL-запит для перевірки синтаксису та семантики на відповідність схемі бази даних. Якщо виявлені помилки, про них повідомляється користувачеві.

2. Оптимізація запитів: оптимізатор запитів оцінює різні плани виконання, вибираючи найефективніший на основі статистики, індексів і оцінок вартості.

3. Виконання плану: вибраний план виконання керує механізмом запитів через послідовність операцій, таких як сканування, фільтрування та об’єднання, для отримання запитаних даних.

4. Доступ до даних: дані отримуються з таблиць за допомогою таких дій, як повне сканування таблиці, сканування індексу або пошук кластерного індексу, залежно від запиту та реалізації даних.

5. Застосування фільтрів: рядки фільтруються відповідно до умов, указаних у реченні WHERE, щоб скоротити проміжні результати на ранній стадії.

6. Об’єднання даних: коли задіяно кілька таблиць, СУБД виконує об’єднання (наприклад, вкладений цикл, хешування або злиття) на основі запиту та доступних індексів.

7. Сортування/групування: якщо вказано, СУБД сортує або групує дані за допомогою ORDER BY, GROUP BY або DISTINCT, використовуючи, де це можливо, індекси для оптимізації продуктивності.

8. Проекція: непотрібні стовпці видаляються, повертаються лише ті, які явно згадуються в списку SELECT.

9. Форматування результату: кінцевий набір результатів форматується та надсилається користувачеві або додатку, завершуючи процес виконання запиту.

В залежності від видів структури запиту, способів реалізації бази даних та інших факторів, деякі етапи виконання можуть бути пропущені або змінені в порядку, залежачи більшою частиною, від оцінки вартості виконання, яка загалом може приймати наступний вигляд:

br – кількість блоків у файлі (блок — це найменша одиниця даних, яку базова система зберігання (hdd, san fs тощо) готова обробляти. Його розмір традиційно становить 512 байт для жорстких дисків.)

hi – позначає висоту індексу

b – кількість блоків, що містять записи з заданим ключем пошуку

n – кількість отриманих записів

Таблиця 1.1 – Вартість різних видів запиту SELECT

Вид SELECT запиту	Вартість	Причини
Лінійний пошук	ts + br * tT	Для цього потрібен один початковий пошук із передачами блоків b r.
Лінійний пошук, рівність за ключем	ts + (br/2) * tT	Це середній випадок, коли потрібен лише один запис, що задовольняє умову. Тому, щойно його буде знайдено, сканування припиняється.
Первинний індекс B + -дерева, рівність за ключем	(hi +1) * (tr + ts)	Кожна операція вводу/виводу потребує одного пошуку та однієї передачі блоку, щоб отримати запис шляхом обходу висоти дерева.
Первинний індекс B + -дерева, рівність на неключі	hi * (tT + ts) + b * tT	Потрібен один пошук для кожного рівня дерева та один пошук для першого блоку.

Продовження таблиці 1.1 – Вартість різних видів запиту SELECT

Вид SELECT запиту	Вартість	Причини
Індекс вторинного B + дерева, рівність за ключем	(hi + 1) * (tr + ts)	Кожна операція вводу/виводу потребує одного пошуку та однієї передачі блоку, щоб отримати запис шляхом обходу висоти дерева.
Вторинний індекс B + -дерева, рівність на неключі	(hi + n) * (tr + ts)	Це вимагає одного пошуку для кожного запису, оскільки кожен запис може бути в іншому блоці.
Первинний індекс B + -дерева, порівняння	hi * (tr + ts) + b * tT	Потрібен один пошук для кожного рівня дерева та один пошук для першого блоку.
Індекс вторинного B + -дерева, порівняння	(hi + n) * (tr + ts)	Це вимагає одного пошуку для кожного запису, оскільки кожен запис може бути в іншому блоці.

Зважаючи на аналіз алгоритмів роботи запитів на вибірку, можна сказати, що маючи таку велику кількість етапів, їх виконання може займати помітну

велику кількість часу, що сильно впливає на ефективність всіх інших пов’язаних елементів компаній та бізнесів, саме тому під час створення будь-яких запитів варто дотримуватись багатьох методів оптимізації запитів. Оптимізація запитів передбачає вибір найефективнішого способу виконання запиту до бази даних за допомогою різноманітних стратегій та інструментів. Далі буде розглянуто **основні методи оптимізації**, які необхідно застосовувати до більшості сучасних та найпоширеніших систем управління базами даних та звертатись до них під час написання кожного з запитів, не тільки на звичайні вибірки даних:

Аналіз плану виконання. Кожен запит, надісланий до бази даних, оптимізатор запитів перетворює на план виконання. У цьому плані описано послідовність операцій, таких як сканування таблиць, пошук індексів або об’єднання, необхідних для отримання запитаних даних. Розуміння того, як аналізувати та інтерпретувати плани виконання допомагає у виявленні етапів, які погано впливають на продуктивність та потенційних покращень.

Індексація. Індекси — це структури даних, які дозволяють швидше отримувати записи, зводячи до мінімуму потребу у повному скануванні таблиці. Ефективні стратегії індексування, такі як індекси з одним стовпцем, складені індекси та індекси покриття, можуть значно підвищити продуктивність запитів. Однак потрібно ретельно керувати компромісом між продуктивністю читання та накладними витратами на підтримку індексів під час оновлення даних.

Об'єднання та стратегії приєднання. Об’єднання таблиць є звичайною операцією в реляційних базах даних. Оптимізатор запитів визначає найкращу стратегію об’єднання на основі таких факторів, як розмір таблиці та доступні індекси. Загальні методи об’єднання включають:

- Об’єднання вкладеного циклу: підходить для невеликих наборів даних, але може працювати повільно з великими таблицями.

- Хеш-об’єднання: ефективне для екві-об’єднань із залученням великих наборів даних.

- Об’єднання злиттям: оптимально для відсортованих наборів даних, що зменшує потребу в додаткових операціях сортування.

Розбиття на розділи та шардинг. Розбиття ділить великий набір даних на менші керовані сегменти на основі певних критеріїв, наприклад діапазонів дат або регіонів. Шардинг розширює цю концепцію для розподілу даних між кількома серверами. Ці методи зменшують обсяг даних, сканованих для запитів, і покращують масштабованість систем.

Використання матеріалізованих представлень. Матеріалізовані представлення зберігають результати попередньо обчислених запитів, що робить наступні запити швидшими завдяки уникненню повторних обчислень. До них варто звертатись при складних агрегацій і часто використовуваних запитів.

Кешування. Кешування тимчасово зберігає результати запиту в пам’яті, зменшуючи необхідність багаторазового виконання того самого запиту. Кешування результатів запитів і кешування проміжних результатів є ефективним у випадках програмах із повторюваними шаблонами доступу.

Управління паралельністю запитів. У багатокористувацьких середовищах виконання одного запиту може вплинути на інші через конкуренцію ресурсів. Такі методи, як пріоритезація запитів, керування робочим навантаженням і належні рівні ізоляції транзакцій, можуть пом’якшити ці проблеми та підтримувати стабільну продуктивність.

Інструменти для оптимізації запитів. Системи керування базами даних забезпечують вбудовані інструменти для допомоги в оптимізації, такі як аналізатори запитів, візуалізатори планів виконання та інформаційні панелі продуктивності, використовуючи їх, можна зекономити багато часу на оптимізації деталей всередині запитів.

Представлені методи можна вважати основами методів оптимізації запитів, їх варто дотримуватись при будь-яких запитах, оскільки ці принципи можна вважати універсальними для широкого спектру звернень до баз даних. Пізніше в роботі будуть розглянуті більш ситуативні методи, які адаптовані до конкретних структур баз даних і робочих навантажень.

1.3 Аналіз видів баз даних

Вибір баз даних є однією з дуже важливим рішення під час створення системи, і очевидно, може помітно вплинути на оптимізації запитів у різних середовищах. Для аналізу було розглянуто найпопулярніші структури, випадки використання та підходи до керування даними. Аналізуючи, враховувались гнучкість моделі даних, популярність і релевантність реальним програмам.

Реляційна база даних (RDBMS). Гарним прикладом реляційної бази даних можна представити PostgreSQL. Ця система керування є однією з найпопулярніших, завдяки своїми можливостям SQL, дотриманням принципів ACID і підтримкою розширених функцій, таких як індексування, віконні функції та матеріалізовані представлення. Окрім PostgreSQL гарними прикладами традиційної RDBMS є MySQL, Microsoft SQL Server, Oracle та Microsoft Access, вони широко використовується в транзакційних системах, що робить їх одним з найкращих варіантів для дослідження методів оптимізації запитів, таких як індексування, перезапис запитів і розділення.

Документно-орієнтовані бази даних. Щоб дослідити модель даних без схеми, можна розглянути MongoDB, як типову базу даних, орієнтовану на документи. Його здатність зберігати дані в документах, подібних до JSON, дозволяє ефективно вміщувати напівструктуровані та неструктуровані дані. Структура індексування та агрегації MongoDB також надає можливість тестувати стратегії оптимізації запитів у середовищі NoSQL.

Магазин ключ-значення. Для сценаріїв, які потребують швидкого пошуку з мінімальними витратами, було обрано Redis. Як сховище ключ-значення в пам’яті Redis оптимізовано для отримання даних із низькою затримкою. Це включення дозволяє досліджувати методи оптимізації, такі як розділення даних і кешування в системах, де простота та швидкість мають пріоритет над складними зв’язками.

База даних родини стовпців. Для аналізу продуктивності колонкового сховища даних було включено Apache Cassandra. Його архітектура, розроблена для високої масштабованості та розподілених систем, підтримує ефективні аналітичні запити до великих наборів даних.

База даних графів. Neo4j було обрано для оцінки представлення та обходу даних на основі графів. Зосереджуючись на взаємозв'язках між точками даних, Neo4j надає інформацію про оптимізацію графових запитів, особливо в сценаріях із рекурсивними або дуже взаємопов'язаними даними.

Тільки після визначення мети роботи та типів даних, які будуть використовуватись, можна обирати вид системи, яка підходить. Реляційні системи та системи NoSQL суттєво відрізняються своїми механізмами зберігання та мовами запитів, у той час як спеціалізовані бази даних, такі як Redis та Neo4j, стосуються нішевих випадків використання. Ця різноманітність дає змогу ідентифікувати методи оптимізації, ефективні в різних парадигмах баз даних, а також ті, що є специфічними для певних моделей даних.

1.4 Види запитів та їх ефективність

У системах баз даних використовуються різні типи запитів залежно від характеру даних і запланованих операцій. Кожен тип запиту відрізняється своїми обчислювальними вимогами та придатністю залежно від структури та розміру бази даних. У цьому розділі класифікуються загальні типи запитів, обговорюються їхні вимоги до ресурсів і надаються рекомендації щодо їх оптимального використання.

Пошукові запити: запити, які отримують певні записи з використанням базових умов фільтрації (SELECT з умовами WHERE). Ресурсомісткість: мінімальна, за умови наявності індексів у відфільтрованих стовпцях.

Рекомендації:

- Використовувати індекси для часто запитуваних полів, щоб скоротити дисковий ввід/вивід.

- Уникати вибору непотрібних стовпців, щоб обмежити передачу даних.

- Підходить для малих і середніх наборів даних і програм реального часу.

Агрегаційні запити: запити, що виконують обчислення, як-от суми, середні значення, підрахунки тощо (GROUP BY із агрегатними функціями). Ресурсомісткість: від помірної до високої, залежно від обсягу даних, які групуються та агрегуються.

Рекомендації:

- Використовувати попередньо агреговані або матеріалізовані подання для великих наборів даних із повторюваними потребами агрегування.

- Розбивати великі набори даних, щоб зменшити обсяг агрегації.

- Використовувати механізми розподілених запитів, такі як Apache Hive або Presto, для середовищ великих даних.

Приєднатися до запитів: запити, що об’єднують дані з кількох таблиць на основі зв’язків (INNER JOIN, LEFT JOIN тощо). Ресурсомісткість: висока, особливо для великих наборів даних без належного індексування або при об’єднанні неключових полів.

Рекомендації:

- Ретельно нормалізовувати дані, щоб мінімізувати непотрібні об’єднання.

- Використовувати індексування стовпців об’єднання, щоб покращити продуктивність.

- Для розподілених баз даних зводити до мінімуму перемішування даних шляхом розміщення пов’язаних наборів даних.

Підзапити та вкладені запити: запити, які містять інший запит (SELECT у SELECT, IN, EXISTS). Ресурсомісткість: змінюється; корельовані підзапити

можуть бути особливо ресурсомісткими, оскільки вони виконуються кілька разів.

Рекомендації:

- Переписувати корельовані підзапити як об’єднання, де це можливо.

- Для повторюваних підзапитів розглядати можливість використання тимчасових таблиць або загальних табличних виразів (CTE).

Оновлення та видалення запитів: запити, які змінюють або видаляють записи (UPDATE, DELETE). Ресурсомісткість: висока для великих таблиць без індексів у цільових стовпцях.

Рекомендації:

- Застосовувати зміни в пакетах для великих наборів даних, щоб уникнути довгих блокувань.

- Індексувати стовпці, які використовуються в реченнях WHERE, щоб прискорити ідентифікацію запису.

- Для кращої ефективності використовувати методи масового оновлення бази даних.

Повнотекстові пошукові запити: запити для отримання текстових даних за допомогою зіставлення шаблонів або пошукових індексів (LIKE, функції повнотекстового пошуку). Ресурсомісткість: висока для великих неіндексованих текстових полів; помірний для індексованих пошукових запитів.

Рекомендації:

- Використовувати індекси повнотекстового пошуку для текстових даних, а не базового зіставлення за зразком.

- Оптимізовувати шаблони запитів, щоб уникнути символів підстановки (%word).

- Для дуже великих наборів даних розглядати зовнішні пошукові системи, такі як Elasticsearch.

Рекурсивні запити: запити, які отримують ієрархічні або самопосилальні структури даних (з використанням WITH RECURSIVE у SQL). Ресурсомісткість: висока, оскільки вони можуть передбачати повторне виконання великих наборів даних.

Рекомендації:

- Обмежувати глибину рекурсії, щоб уникнути надмірних обчислень.
- Розглядати альтернативні представлення, такі як списки суміжності або вкладені набори, для ефективнішого обходу.
- Використовувати попередньо обчислені шляхи для статичних ієрархічних даних.

1.5 Додаткова оптимізація запитів

Окрім базових методів, розширені методи оптимізації запитів можуть значно підвищити продуктивність, особливо для складних систем або середовищ із високим попитом. У цьому розділі розглядаються додаткові стратегії з простими прикладами для демонстрації їх застосування. Список відсортований від найважливіших методів, які має сенс використовувати у більшості випадків, до більш ситуативних, які необхідно дотримуватись при великих обсягах даних.

Оптимізація операторів SELECT. Під час виконання запиту SELECT *, базі даних доводиться отримувати всі стовпці таблиці, навіть якщо деякі з них не потрібні. Уточнення необхідних стовпців скорочує обсяг переданих даних, зменшуючи навантаження на мережу та час обробки результатів.

Неефективний запит:

```
SELECT * FROM employees;
```

Оптимізація: замість вказування на отримання повного списку всіх стовпців з таблиці, використовуючи символ “*”, можна вказувати лише ті стовпці, які є актуальними тільки для цієї ситуації:

SELECT name, salary FROM employees;

Завдяки отриманню лише відповідних стовпців запит зменшує витрати на передачу даних і обробку.

Пакетна обробка.

Виконання багатьох запитів окремо (наприклад, вставки) викликає кожен раз нове з'єднання з сервером. Групування запитів в один пакет дозволяє зменшити кількість таких з'єднань, що значно знижує затримки.

Неефективний запит: багаторазове вставлення записів:

INSERT INTO sales (id, amount) VALUES (1, 100);

INSERT INTO sales (id, amount) VALUES (2, 200);

Оптимізація: виділення значень, які вносяться до таблиці в групу та занесення всіх значень водночас, використовуючи одинарний запуск функції:

INSERT INTO sales (id, amount) VALUES

(1, 100),

(2, 200);

Використання тимчасових таблиць.

Складні запити, які включають кілька об’єднань і умов, можна поділяти на частини, попередньо зберігши проміжні результати у тимчасових таблицях. Це значно зменшує кількість повторних обчислень та суттєво зменшує час виконання запиту.

Складний запит:

SELECT p.name, SUM(s.amount)

FROM products p

JOIN sales s ON p.product_id = s.product_id

WHERE s.date > '2024-01-01'

GROUP BY p.name;

Оптимізація: визначення даних, які часто використовуються, або розрахунків до яких часто звертаються (в наявному прикладі це будуть результати пошуку, які мають дати більше “2024-01-01”) та внесення цих даних до тимчасової таблиці, потім замість повторних розрахунків, можна звернутись до нової таблиці та взяти значення з неї:

```
CREATE TEMPORARY TABLE filtered_sales AS
SELECT * FROM sales WHERE date > '2024-01-01';
SELECT p.name, SUM(fs.amount)
FROM products p
JOIN filtered_sales fs ON p.product_id = fs.product_id
GROUP BY p.name;
```

Індексні підказки.

У деяких випадках оптимізатор запитів в базі даних може обрати не найефективніший індекс. Явне вказування індексу спрощує обчислення, особливо у великих таблицях.

Приклад: застосоване вказування індексу через функцію INDEX(), потім до якої виконується порівняння “<100”:

```
SELECT * FROM products WITH (INDEX(idx_price)) WHERE price < 100;
```

Це гарантує, що оптимізатор використовує вказаний індекс для кращої продуктивності.

Денормалізація для запитів, що інтенсивно читають.

Денормалізація включає дублювання даних у таблиці для зменшення кількості об’єднань під час виконання складних запитів. Це дозволяє знизити час виконання та зменшити навантаження на сервер, особливо у великих системах, де відбувається багаторазовий доступ до одних і тих самих даних. Вона збільшує продуктивність у сценаріях з інтенсивним читанням.

Нормалізований запит схеми:

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

Оптимізація: внесення даних зі стовпця customer_name таблиці customers до головної таблиці замовлень orders для швидкого пошуку інформації:

SELECT order_id, customer_name FROM orders;

Уникнення функцій на індексованих стовпцях.

Використання функцій на індексованих стовпцях робить індекси недієздатними, оскільки вони не враховуються у плані виконання запиту. Наприклад, замість YEAR(order_date) = 2024 слід використовувати умову з діапазоном BETWEEN, що дозволяє використовувати індекс і скоротити час пошуку даних у великих таблицях.

Неефективний запит:

SELECT * FROM orders WHERE YEAR(order_date) = 2024;

Оптимізація: переписання запиту, уникаючи можливі функції до будь-яких індексованих стовпців, у наданому прикладі - функція YEAR була застосована до стовпця order_date, що могло викликати проблеми та затримки, замість неї можна використовувати простий діапазон значень, які відповідають тим самим результатам, наприклад використовуючи оператор BETWEEN:

SELECT * FROM orders WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31';

Планування запитів.

При існуванні запитів, які займають дуже багато часу, або викликають велику нагрузку на сервери (напр. копіювання БД, створення точки відновлення), варто розглядати варіанти використання вбудованих інструментів для планування запитів, щоб запускати їх у непіковий час, наприклад вночі, коли навантаження мінімальне.

Приклад: Виконання запиту на агрегацію великої кількості даних кожен день о 23:00:00:

```
CREATE EVENT daily_aggregation
ON SCHEDULE EVERY 1 DAY STARTS '2024-12-10 23:00:00'
DO
INSERT INTO sales_summary (day, total_sales)
```

SELECT order_date, SUM(amount)

FROM sales GROUP BY order_date;

Паралелізм запитів.

Паралельне виконання запитів розподіляє завдання між кількома процесами, використовуючи багатоядерну архітектуру сучасних серверів. Це корисно для великих наборів даних, що вимагають обчислень, наприклад, під час агрегації.

Приклад (PostgreSQL): зміна конфігурації max_parallel_workers_per_gather в СУБД з метою використання паралельності запитів:

SET max_parallel_workers_per_gather = 4;

SELECT SUM(sales) FROM transactions WHERE date > '2024-01-01';

Попереднє обчислення результатів за допомогою індексованих переглядів.

Створення матеріалізованих переглядів дозволяє зберігати результати складних запитів у таблицях, що індексуються. Наприклад, результати підсумовування продажів можуть бути збережені у попередньо обчисленій таблиці, що значно скорочує час виконання повторних запитів.

Приклад: створення матеріалізованого представлення sales_summary для майбутнього використання, який вже буде включати готові результати розрахунків суми кількості продуктів (SUM(amount)):

CREATE VIEW sales_summary AS

SELECT product_id, SUM(amount) AS total_sales

FROM sales GROUP BY product_id

WITH SCHEMABINDING;

CREATE UNIQUE CLUSTERED INDEX idx_sales_summary

ON sales_summary(product_id);

Профілювання та моніторинг запитів.

Інструменти профілювання, такі як EXPLAIN у MySQL або Live Query Statistics у SQL Server, дозволяють аналізувати план виконання запитів, виявляти

етапи, які використовують більше всього ресурсів та оптимізувати запит, знаючи проблеми. Це забезпечує повну прозорість і контроль під час оптимізації.

Приклад інструменту: у СУБД MySQL інструмент для монуторингу запитів викликається функцією EXPLAIN, у прикладі показано використання цієї функції для отримання даних по пошуковому запиту з фільтруванням по даті:

```
EXPLAIN SELECT * FROM orders WHERE order_date > '2024-01-01';
```

1.6 Аналіз досліджень з оптимізації запитів до баз даних

Ще до визначення теми цієї роботи та під час її виконання було розглянуто багато аналогічних робіт, які ставили собі за мету оптимізувати роботу запитів до баз даних**.** У цьому розділі присутні результати аналізу досліджень та виокремлення частин, які відсутні або недостатньо обговорені, у порівнянні з інформацією в цій дипломній роботі:

Стаття [5] розглядаються проблеми, пов’язані з обробкою великих запитів до бази даних, і представлена інноваційна методологія оптимізації. У дослідженні підкреслюється, що зі збільшенням розмірів бази даних звичайні стратегії виконання запитів демонструють зниження ефективності, навіть із вдосконаленням апаратних можливостей.

В цій роботі пропонують техніку підвищення продуктивності, що передбачає заміну оператора SQL IN тимчасовими таблицями та застосування некластеризованих індексів. Такий підхід мінімізує логічні операції, тим самим покращуючи швидкість виконання запитів і загальну ефективність.

Експериментальні результати показують, що запропонований метод підвищує продуктивність запитів на 15% для малих баз даних (10 ГБ) і на 17% для баз даних середнього розміру (50 ГБ). У документі міститься комплексний огляд існуючих стратегій оптимізації та наголошується на необхідності

збалансування продуктивності системи з доступністю для користувачів. Автори роблять висновок, що впровадження таких методологій може значно підвищити ефективність систем управління реляційними базами даних при обробці великомасштабних наборів даних.

Це дослідження є достатньо корисним для вирішення проблем, пов'язаних з ефективністю операторів IN в SQL, в статті рекомендують деякі способи оптимізації, які були були вказані в цій роботі, наприклад використання тимчасових таблиць, але головним недоліком є те, що представлений тільки одна проблема, з яку можна навіть не отримати до використання дуже великих баз даних.

У дослідницькій статті [6] порівнюється продуктивність двох популярних баз даних NoSQL: MongoDB і CouchDB. В роботі досліджується, як швидко кожна база даних може виконувати типові операції, такі як додавання нових даних, видалення даних, оновлення даних і отримання даних. Щоб забезпечити чесне порівняння, дослідники використали ідентичні набори даних і запити для обох баз даних, а потім виміряли час, необхідний для виконання кожної операції.

Результати показали, що MongoDB загалом перевершує CouchDB у більшості завдань. MongoDB була значно швидшою при додаванні великих обсягів даних, тоді як CouchDB продемонструвала трохи кращу продуктивність в оновленні даних.

У висновку вказано, що MongoDB добре підходить для програм, які вимагають високої швидкості виконання, що робить її ідеальним вибором для багатьох великих корпоративних програм. І навпаки, сильні сторони CouchDB у конкретних сценаріях, таких як оновлення даних, можуть зробити його більш придатним варіантом для невеликих або спеціалізованих проектів.

Обрана дослідницька стаття прекрасно описує швидкість використання запитів до різних СУБД, які використовують структуру NoSQL, результати гарно вказують на те, що правильний вибір системи управління може значно вплинути на ефективність запитів, але на жаль в ній майже відсутні прямі методи оптимізації запитів, звертаючи увагу тільки на сумісність.

В роботі [7] розглянуто методи оптимізації запитів, зосереджуючись на практичних аспектах підвищення продуктивності систем. Основна увага приділяється таким інструментам, як аналіз планів виконання запитів (Execution Plans), вбудованим засобам профілювання продуктивності, оптимізації індексів, а також використанню механізмів кешування.

Fritchey наголошує на важливості розуміння планів виконання для діагностики проблем із продуктивністю. Він пропонує конкретні стратегії, зокрема створення покриваючих індексів (covering indexes), уникнення витратних операцій, таких як сканування всього індексу (index scan), та використання рекомендацій щодо оптимального запису запитів. Автор також пояснює, як працювати з паралельними запитами та розподілом навантаження для великих баз даних.

Проте книга зосереджена лише на SQL Server, не аналізуючи інших СУБД і альтернативних структур даних, що є її обмеженням. У цьому дипломному дослідженні використовуються деякі ідеї, представлені Fritchey, але вони поєднуються з більш ширшим підходом. Зокрема, увага приділяється порівнянню методів оптимізації різних типів баз даних і адаптації підходів залежно від їхньої структури.

У цьому дослідженні [8] автор порівнює продуктивність різних систем керування базами даних (СКБД), зокрема MySQL, PostgreSQL та MongoDB. Основний акцент зроблено на вимірюванні часу виконання базових операцій (вставка, видалення, пошук), ресурсомісткість запитів і ефективність роботи із різними обсягами даних. Особливу увагу приділено аналізу реляційних та нереляційних баз даних, враховуючи специфіку їх архітектури.

Дослідження показало, що MySQL є ефективним рішенням для невеликих транзакційних систем завдяки стабільній роботі із структурованими даними. PostgreSQL, своєю чергою, демонструє найкращі результати у складних аналітичних запитах завдяки підтримці розширених функцій, таких як віконні функції та матеріалізовані представлення. MongoDB виділяється високою

швидкістю роботи з неструктурованими та напівструктурованими даними, що робить його придатним для систем, орієнтованих на гнучкість.

У порівнянні з цією роботою, дипломна робота має ширший акцент: замість аналізу конкретних СУБД, досліджено універсальні підходи до оптимізації запитів, незалежно від типу бази даних. Крім того, розглядено не лише ефективність окремих операцій, а й комплексні стратегії оптимізації, такі як нормалізація, денормалізація, індексація та кешування.

У своїй роботі [9] Меліса Голлінгсворт досліджує важливість нормалізації та денормалізації даних як ключових інструментів оптимізації баз даних. Автор пояснює, що нормалізація – це процес структурування бази даних таким чином, щоб усунути надмірність даних і забезпечити цілісність. Зокрема, основна мета нормалізації полягає у мінімізації дублювання даних та підвищенні ефективності оновлення. Голлінгсворт описує кілька форм нормалізації (від першої до п’ятої) та ілюструє їхні переваги для великих систем із динамічною модифікацією даних.

Разом з тим, автор також акцентує увагу на денормалізації – зворотному процесі, коли деяке дублювання даних вводиться для підвищення швидкості запитів. Денормалізація, за словами автора, є оптимальним рішенням для баз даних, що активно використовуються для читання, оскільки зменшує кількість об’єднань таблиць під час виконання запитів. Голлінгсворт наголошує на важливості правильного балансу між нормалізацією та денормалізацією, залежно від характеру застосування бази даних.

На відміну від її роботи, ця дипломна робота зосереджується на практичному тестуванні цих методів. Ми аналізуємо їх вплив на продуктивність, враховуючи різні структури баз даних (реляційні та NoSQL), та розширюємо підхід шляхом інтеграції додаткових стратегій, таких як кешування та шардинг. Ця практична оцінка дозволяє зробити висновки більш адаптивними до різноманітних сценаріїв використання.

Ця праця [10] присвячена проектуванню реляційних баз даних із акцентом на структурованому підході до створення високоефективних систем. Основна

увага приділяється принципам нормалізації, використанню індексів і створенню реляційних моделей, які відповідають потребам різних бізнес-додатків. Автори акцентують на важливості балансування між нормалізацією та денормалізацією залежно від типів навантаження на базу даних. Особлива увага приділяється проєктуванню ключів (первинних, зовнішніх та унікальних), які забезпечують цілісність даних і полегшують доступ до них.

У книзі також розглянуто питання управління продуктивністю баз даних, такі як побудова складених індексів і оптимізація JOIN-запитів. Приділено увагу структурі таблиць, їх відповідності нормальним формам та методам зниження надмірностей даних для великих баз.

Головна відмінність аналогу від цієї роботи полягає у її зосередженості виключно на реляційних базах даних, таких як SQL Server, та їх проектуванні. У цій роботі враховано також нереляційні бази (NoSQL), такі як MongoDB та Cassandra, і запропоновані рішення для оптимізації запитів у змішаних середовищах. Додатково, протестовано методи оптимізації не лише на рівні проєктування, але й у реальному виконанні запитів з використанням сучасних інструментів аналізу продуктивності.

Після аналізу наданих робіт та багатьох інших, які не були сказані, можна зробити висновок, що більшість робіт, які стосуються тематики оптимізації запитів до баз даних мають такий список недоліків:

- є застарілими, багато методів мають кращі сучасні аналоги, або перестали бути актуальними через покращення роботи СУБД;

- вузько-направленними, стосуються тільки однієї СУБД або типів структури;

- мають тільки теоретичну частину без практичних порівнянь результатів оптимізації;

- надають відносно невелику кількість методів оптимізації.

Саме тому й було вирішено написати дипломну роботу на цю тему та виправити більшість цих недоліків в одній скомпонованій праці.

2 ПРАКТИЧНА ЧАСТИНА

2.1 Підготовка до практичної частини та створення набору даних

Практична частина включає в себе створення тестової бази даних, в якій будуть виконуватись різні види запитів з урахуванням вказаних раніше методів оптимізації. Зважаючи на досвід у роботі з БД, мною було обрано використовувати Microsoft SQL Server — це власна система керування реляційними базами даних, розроблена корпорацією Майкрософт, яка є однією з найпоширеніших виборів у великих компаніях. Для спрощення роботи також було використано програмний додаток SQL Server Management Studio (SSMS), який використовується для налаштування, керування та адміністрування всіх компонентів у Microsoft SQL Server.

Інформація про ПК, на якому виконувалась робота:

Процесор: AMD Ryzen 5 7500F 6-Core Processor 3.70 GHz

ОЗП: 32 Гб

ОС: Windows 10 Pro, 64-розрядна операційна система

Диск: KINGSTON SNV2S1000G, Місткість:932 ГБ

Перше, що було зроблене, це створення таблиць для імен FirstNames та прізвищ LastNames, які пізніше були використані для заповнення основних таблиць, було вирішено занести по 50 значень до кожної з таблиць

Створювались таблиці з допомогою запитів CREATE TABLE:

```
CREATE TABLE LastNameTable (
   NameID INT PRIMARY KEY,
   LastName NVARCHAR(50)
);
```

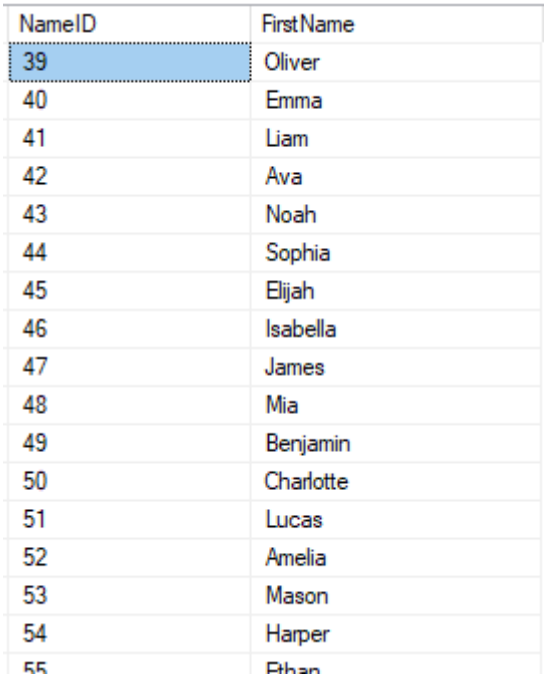
Потім використовуючи стандартний запит INSERT INTO було занесено дані:

```
INSERT INTO FirstNames (FirstName)
```

VALUES ('Oliver'), ('Emma'), … , ('Hazel');

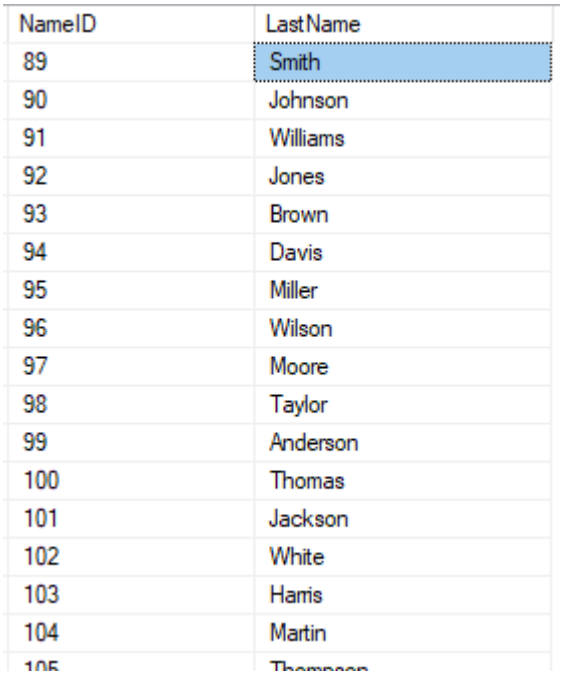
INSERT INTO LastNames (LastName)

VALUES ('Smith'), ('Johnson'), … , ('Harrison');



NameID	FirstName
39	Oliver
40	Emma
41	Liam
42	Ava
43	Noah
44	Sophia
45	Elijah
46	Isabella
47	James
48	Mia
49	Benjamin
50	Charlotte
51	Lucas
52	Amelia
53	Mason
54	Harper
55	Ethan

Рисунок 2.1 – Частина записів у створеній таблиці FirstNames



NameID	LastName
89	Smith
90	Johnson
91	Williams
92	Jones
93	Brown
94	Davis
95	Miller
96	Wilson
97	Moore
98	Taylor
99	Anderson
100	Thomas
101	Jackson
102	White
103	Harris
104	Martin
105	Thompson

Рисунок 2.2 – Частина записів у створеній таблиці LastNames

Для основної таблиці було обрано 9 змінних у вигляді ненормалізованої бази даних. Варто відмітити, що для більших баз, майже завжди бажано виконувати розбиття на розділи та шардінг, де це є можливим, оскільки це значно полегшує майбутнє масштабування та подальшу роботу з запитами. Для заповнення таблиці були використані дві попередні таблиці та додатково 20 різних країн, для отримання більш явних результатів оптимізації було вирішено заповнити таблицю 500 000 унікальними рядками.

```
CREATE TABLE Users (
    UserID INT IDENTITY(1,1) PRIMARY KEY,
    FirstNameId VARCHAR(100),
    LastNameId VARCHAR(100),
    Email VARCHAR(100) UNIQUE,
    DateOfBirth DATE,
    CountryId VARCHAR(100),
    SignupDate DATETIME,
    PhoneNumber VARCHAR(15),
    IsActive BIT DEFAULT 1,
    LastLoginDate DATETIME
);
```

Рисунок 2.3 – Текст запиту для створення головної таблиці

В структурі існуючої бази даних, створені таблиці знаходяться за однієї директорією, для більш зручного використання.

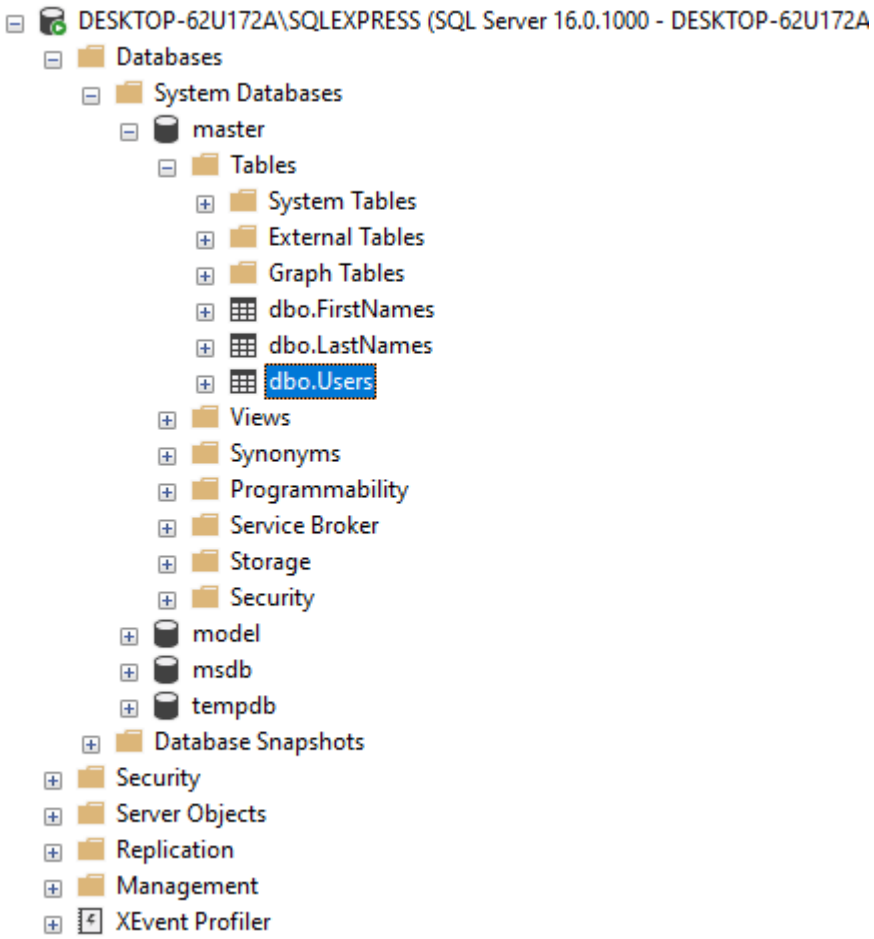


Рисунок 2.4 – Розміщення та загальна структура бази даних

Заповнення таблиці Users було виконано такими методами:

FirstName та LastName – внесення даних з існуючих таблиць;

Email – значення складається з імені та прізвища з унікальним індексом та доменом;

DateOfBirth, SignUpDate, LastLoginDate – випадкове значення в обраних діапазонах, використовуючи функцію DATEADD;

IsActive – випадкове бінарне значення 1 або 0, з ймовірністю 9 до 1, відповідно;

PhoneNumber – поєднання префіксу “+1” з випадковим 10-значним числом;

Country – випадкове значення з наданого списку.

```sql
SET NOCOUNT ON;

DECLARE @i INT = 1;
DECLARE @maxRows INT = 500000;

WHILE @i <= @maxRows
BEGIN
    INSERT INTO Users (FirstName, LastName, Email, DateOfBirth, Country, SignupDate, PhoneNumber, IsActive, LastLoginDate)
    SELECT
        FN.FirstName,
        LN.LastName,
        CONCAT(FN.FirstName, '.', LN.LastName, @i, '@example.com'), -- Unique email
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 20000), GETDATE()), -- Random past date
        CASE (ABS(CHECKSUM(NEWID()) % 5) + 1)
            WHEN 1 THEN 'Ukraine'
            WHEN 2 THEN 'UK'
            WHEN 3 THEN 'Canada'
            WHEN 4 THEN 'Australia'
            WHEN 5 THEN 'India'
            WHEN 6 THEN 'Germany'
            WHEN 7 THEN 'France'
            WHEN 8 THEN 'Italy'
            WHEN 9 THEN 'Spain'
            WHEN 10 THEN 'Brazil'
            WHEN 11 THEN 'Japan'
            WHEN 12 THEN 'South Korea'
            WHEN 13 THEN 'China'
            WHEN 14 THEN 'USA'
            WHEN 15 THEN 'Mexico'
            WHEN 16 THEN 'South Africa'
            WHEN 17 THEN 'Argentina'
            WHEN 18 THEN 'Netherlands'
            WHEN 19 THEN 'Sweden'
            WHEN 20 THEN 'Norway'
        END,
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 1000), GETDATE()), -- Random past signup date
        CONCAT('+1', RIGHT(ABS(CHECKSUM(NEWID())), 10)), -- Random phone number
        CASE WHEN @i % 10 = 0 THEN 0 ELSE 1 END, -- Randomly set IsActive (inactive for every 10th user)
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 365), GETDATE())
    FROM
        (SELECT TOP 1 FirstName FROM FirstNames ORDER BY NEWID()) FN
    CROSS APPLY
        (SELECT TOP 1 LastName FROM LastNames ORDER BY NEWID()) LN;

    SET @i += 1;
END;

SET NOCOUNT OFF;
```

Рисунок 2.5 – Текст запиту для заповнення головної таблиці Users

UserID	FirstName	LastName	Email	DateOfBirth	Country	SignupDate	PhoneNumber	IsActive	LastLoginDate
22	Isaiah	Smith	Isaiah.Smith2@example.com	1990-05-21	Australia	2024-04-06 21:19:00.123	+1577375094	1	2024-01-30 21:19:00.123
23	James	Perez	James.Perez3@example.com	1975-11-05	Canada	2024-10-01 21:19:00.123	+11340040257	1	2024-01-27 21:19:00.123
24	Charlotte	White	Charlotte.White4@example.com	2005-01-14	India	2024-05-18 21:19:00.123	+1328239047	1	2024-01-07 21:19:00.123
25	Aria	Williams	Aria.Williams5@example.com	1992-04-08	Ukraine	2022-06-29 21:19:00.123	+1836272908	1	2024-10-24 21:19:00.123
26	Elijah	Murphy	Elijah.Murphy6@example.com	2015-07-05	Canada	2022-06-26 21:19:00.123	+11047021862	1	2024-07-18 21:19:00.123
27	Harper	Williams	Harper.Williams7@example.com	2000-05-28	UK	2023-05-05 21:19:00.123	+1225674706	1	2024-10-18 21:19:00.123
28	Evelyn	Murphy	Evelyn.Murphy8@example.com	2003-06-19	Ukraine	2022-06-23 21:19:00.123	+12118138623	1	2024-10-28 21:19:00.123
30	Madison	Moore	Madison.Moore10@example.com	1994-09-11	India	2023-06-10 21:19:00.127	+11527518032	0	2024-10-28 21:19:00.127
31	Isaiah	Roberts	Isaiah.Roberts11@example.com	2005-08-19	UK	2023-04-25 21:19:00.127	+11176089819	1	2024-09-03 21:19:00.127
34	Michael	Miller	Michael.Miller14@example.com	1973-05-30	Ukraine	2024-04-07 21:19:00.127	+1180794895	1	2024-07-28 21:19:00.127
35	Mia	Bailey	Mia.Bailey15@example.com	1996-06-06	Ukraine	2023-02-12 21:19:00.127	+1988318311	1	2024-10-16 21:19:00.127

Рисунок 2.6 – Занесені записи до таблиці Users

2.2 Тестові сценарії та порівняння результатів

Далі в роботі будуть розглянуті методи оптимізації, результати яких можна продемонструвати на створеній базі даних, для отримання необхідних даних буде використано вбудований інструментарій “**Live Query Statistics**” [10] та

“**Execution Plan**”, головними параметрами для вимірювання було визначено час виконання головного оператора та загальний час виконання запиту:

Оптимізація операторів SELECT (вибірка тільки необхідних даних, замість виклику всієї таблиці)

Приклад завдання запиту: отримати імена та прізвища всіх користувачів, дата народження яких більша за 19.02.2001

```
SELECT [UserID]
      ,[FirstName]
      ,[LastName]
      ,[Email]
      ,[DateOfBirth]
      ,[Country]
      ,[SignupDate]
      ,[PhoneNumber]
      ,[IsActive]
      ,[LastLoginDate]
  FROM [master].[dbo].[Users]
WHERE DateOfBirth > '2001-02-19'
```

Рисунок 2.7 – Тестовий неоптимізований запит з умовою WHERE

Вибірка результатів вимірювання запитів:

Таблиця 2.1 – Вибірка часу виконання запиту

Index Scan (s)	,163	,125	,162	,159	,162	,124	,159	,124	,124	,124
Total time (s)	,127	,137	,138	,122	,136	,140	,134	,134	,133	,216

Середній час оператора: 0,142 с (стандартне відхилення: 0,018 с).

Середній загальний час виконання: 1,142 с (стандартне відхилення: 0,025 с).

Попередньо варто відмітити, що перший запуск будь-якого запиту виконується довше, оскільки під час нього створюється план виконання, у нашому випадку перше використання оператора зайняло 0,518 с. Для більш

стабільного аналізу та збору даних, перший запуск не буде входити до вимірювання.

Оптимізація полягає в зменшенні кількості стовпців, які оператор SELECT буде використовувати, наприклад, якщо ми будемо використовувати тільки стовпці ідентифікаторів користувачів, їх ім’я та прізвище, то варо вказувати тільки їх:

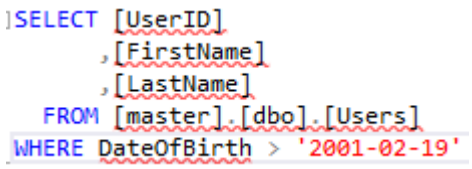


Рисунок 2.8 – Тестовий оптимізований запит з умовою WHERE

Вибірка результатів вимірювання запитів:

Таблиця 2.2 – Вибірка часу виконання запиту

Index Scan (s)	,068	,070	,070	,077	,067	,067	,068	,069	,068	,069
Total time (s)	,591	,604	,591	,597	,598	,591	,589	,595	,592	,590

Середній час оператора: 0,069 с (стандартне відхилення: 0,003 с).

Середній загальний час виконання: 0,594 с (стандартне відхилення: 0,004 с).

Просте видалення зайвої інформації з результатів пришвидшило виконання в два рази, як у випадку часу оператора, так і загалом, що сильно може вплинути в запитах, де є багаторазові або циклічні звернення до таблиць.

Нормалізація бази даних.

Нормалізація баз даних дозволяє, в залежності від розмірів таблиць та їх структур зекономити значну частину простору на серверах і, що більш важливо,

дозволяють швидке використання окремих частин баз даних та таблиць в інших запитах, без необхідності задіювання всієї великої таблиці.

Для експерименту було створено таблицю NormUsers, яка мала також 500 000 рядків, але тепер, замість стовпців зі значеннями імен, прізвищ та країн, були вказані відповідні ідентифікатори до інших таблиць.

```
CREATE TABLE NormUsers (
    UserID INT IDENTITY(1,1) PRIMARY KEY,
    FirstNameId INT,
    LastNameId INT,
    Email VARCHAR(100) UNIQUE,
    DateOfBirth DATE,
    CountryId INT,
    SignupDate DATETIME,
    PhoneNumber VARCHAR(15),
    IsActive BIT DEFAULT 1,
    LastLoginDate DATETIME
);
```

Рисунок 2.9 – Текст запиту для створення таблиці в нормалізованій системі

Запит для заповнення таблиці NormUsers загалом має ідентичну структуру до таблиці Users, головні відмінності полягають у стовцях FirstNameId, LastNameId, CountryId, які заповнювались випадковими значеннями від 1 до 50, використовуючи функцію RAND.

```
SET NOCOUNT ON;

DECLARE @i INT = 1;

WHILE @i <= 500000
BEGIN
    INSERT INTO NormUsers (FirstNameId, LastNameId, Email, DateOfBirth, CountryId,
        SignupDate, PhoneNumber, IsActive, LastLoginDate)
    VALUES (
        (RAND() * 49 + 1),
        (RAND() * 49 + 1),
        CONCAT('exampleEmail', @i, '@example.com'),
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 20000), GETDATE()), -- Random past date
        (RAND() * 19 + 1),
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 1000), GETDATE()), -- Random past signup date
        CONCAT('+38', RIGHT(ABS(CHECKSUM(NEWID())), 10)), -- Random phone number
        CASE WHEN @i % 10 = 0 THEN 0 ELSE 1 END, -- Randomly set IsActive (inactive for every 10th user)
        DATEADD(DAY, -ABS(CHECKSUM(NEWID()) % 365), GETDATE())
    );
SET @i += 1;
END;

SET NOCOUNT OFF;
```

Рисунок 2.10 – Текст запиту для заповнення таблиці NormUsers

Після створення обох видів структур баз даних, можна за допомогою вбудованих інструментів порівняти властивості таблиць, у нашому випадку, головною характеристикою буде кількість місця, яке вони займають.

Ненормалізована система: 56,961 МБ.

Нормалізована система: 52,641 МБ.

Compression	
Compression type	None
Filegroups	
FILESTREAM filegroup	
Table is partitioned	False
Text filegroup	
Filegroup	PRIMARY
General	
Data space	56,961 MB
Vardecimal storage format is enabled	False
Index space	49,625 MB
Row count	500000

Рисунок 2.11 – Властивості таблиці в ненормалізованій системі (57 Мб)

Compression	
Compression type	None
Filegroups	
FILESTREAM filegroup	
Table is partitioned	False
Text filegroup	
Filegroup	PRIMARY
General	
Data space	52,641 MB
Vardecimal storage format is enabled	False
Index space	44,477 MB
Row count	500000

Рисунок 2.12 – Властивості таблиці в нормалізованій системі (52,6 Мб)

Тепер можна порівняти швидкість виконання запитів до систем з різними структурами. Було використано звичайний запит SELECT для отримання всіх значень з таблиць. У випадку ненормалізованої таблиці просто було отримані всі її значення.

```
SELECT [UserID]
      ,[FirstName]
      ,[LastName]
      ,[Email]
      ,[DateOfBirth]
      ,[Country]
      ,[SignupDate]
      ,[PhoneNumber]
      ,[IsActive]
      ,[LastLoginDate]
  FROM [master].[dbo].[Users]
```

Results | Messages | Live Query Statistics | Execution plan

UserID	FirstName	LastName	Email	DateOfBirth	Country	SignupDate	PhoneNumber	IsActive	LastLoginDate
21	Amelia	Anderson	Amelia.Anderson1@example.com	2018-03-03	NULL	2023-07-02 21:19:00.123	+11189191611	1	2024-10-22 21:19:00.123
22	Isaiah	Smith	Isaiah.Smith2@example.com	1990-05-21	Australia	2024-04-06 21:19:00.123	+1577375094	1	2024-01-30 21:19:00.123
23	James	Perez	James.Perez3@example.com	1975-11-05	Canada	2024-10-01 21:19:00.123	+11340040257	1	2024-01-27 21:19:00.123
24	Charlotte	White	Charlotte.White4@example.com	2005-01-14	India	2024-05-18 21:19:00.123	+1328239047	1	2024-01-07 21:19:00.123
25	Aria	Williams	Aria.Williams5@example.com	1992-04-08	Ukraine	2022-06-29 21:19:00.123	+1836272908	1	2024-10-24 21:19:00.123
26	Elijah	Murphy	Elijah.Murphy6@example.com	2015-07-05	Canada	2022-06-26 21:19:00.123	+11047021862	1	2024-07-18 21:19:00.123
27	Harper	Williams	Harper.Williams7@example.com	2000-05-28	UK	2023-05-05 21:19:00.123	+1225674706	1	2024-10-18 21:19:00.123
28	Evelyn	Murphy	Evelyn.Murphy8@example.com	2003-06-19	Ukraine	2022-06-23 21:19:00.123	+12118138623	1	2024-10-28 21:19:00.123
29	Harper	Martinez	Harper.Martinez9@example.com	2014-06-25	NULL	2022-06-08 21:19:00.127	+1615512118	1	2024-12-04 21:19:00.127

Рисунок 2.13 – Тестовий запит, який звертається до ненормалізованої БД

Використовуючи вбудовані інструменти Execution Plan у нас є можливість подивитись на план виконання, якому слідував запит, утворені зв’язки та головні використані функції.

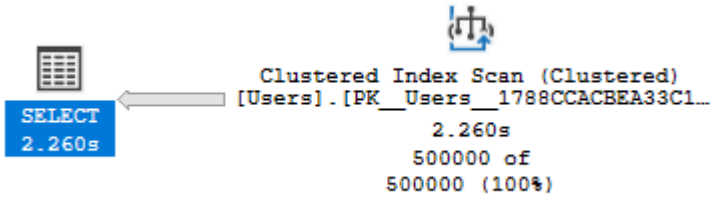


Рисунок 2.14 – План виконання запиту у ненормалізованій БД

Вибірка результатів вимірювання запитів:

Таблиця 2.3 – Вибірка часу виконання запиту

Index Scan (s)	,228	,230	,229	,228	,228	,227	,229	,228	,230	,230
Total time (s)	,496	,538	,513	,507	,498	,500	,495	,497	,506	,504

Середній час оператора: 0,229 с (стандартне відхилення: 0,001 с).

Середній загальний час виконання: 2,505 с (стандартне відхилення: 0,012 с).

Для отримання значень в нормалізованій системі, необхідно додатково вказати зв’язок між пов’язаними таблицями імен та прізвищ через JOIN.

```
SELECT
    [UserID]
    ,fn.firstname
    ,ln.lastname
    ,[Email]
    ,[DateOfBirth]
    ,[CountryId]
    ,[SignupDate]
    ,[PhoneNumber]
    ,[IsActive]
    ,[LastLoginDate]
FROM [master].[dbo].[NormUsers] nu
    inner join firstnames fn on nu.firstnameid = fn.nameid
    inner join lastnames ln on nu.lastnameid = ln.nameid
```

Results | Messages | Live Query Statistics | Execution plan

UserID	firstname	lastname	Email	DateOfBirth	CountryId	SignupDate	PhoneNum
1	Charlotte	Collins	exampleEmail1@example.com	2004-09-10	17	2023-04-04 23:35:31.343	+3814785
2	Grace	Roberts	exampleEmail2@example.com	2005-02-06	19	2024-01-24 23:35:31.343	+3819379
3	Jack	Wilson	exampleEmail3@example.com	1994-07-08	19	2022-06-29 23:35:31.343	+3860646
4	Daniel	Perez	exampleEmail4@example.com	1973-09-28	5	2022-07-21 23:35:31.343	+3868778
5	Penelope	Baker	exampleEmail5@example.com	2005-09-29	19	2023-01-03 23:35:31.343	+3815938
6	Ethan	Hughes	exampleEmail6@example.com	1995-08-24	9	2024-04-11 23:35:31.343	+3878622
7	Daniel	Gomez	exampleEmail7@example.com	2013-02-28	8	2024-12-09 23:35:31.343	+3896649
8	Avery	Thompson	exampleEmail8@example.com	1995-04-28	6	2024-11-01 23:35:31.343	+3877101
9	Scarlett	Adams	exampleEmail9@example.com	2014-06-02	12	2023-09-04 23:35:31.343	+3861717
10	Alexander	Martinez	exampleEmail10@example.com	2004-07-04	16	2022-04-19 23:35:31.343	+3899552

Рисунок 2.15 – Тестовий запит, який звертається до нормалізованої БД

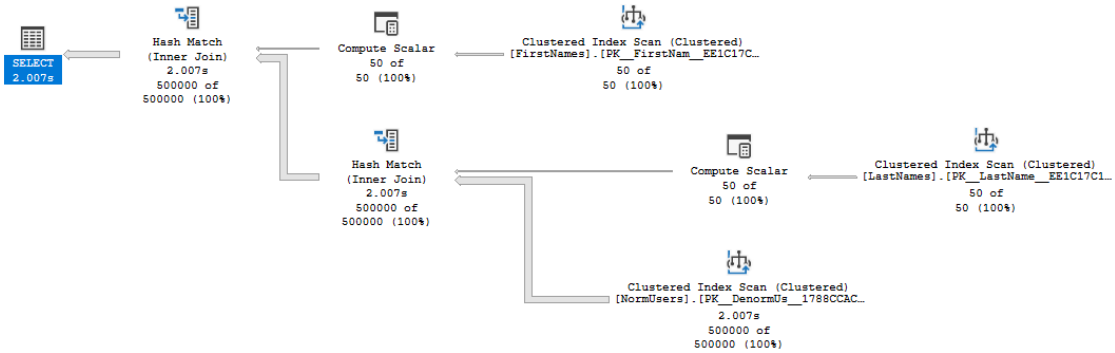


Рисунок 2.16 – План виконання запиту у нормалізованій БД

Вибірка результатів вимірювання запитів:

Таблиця 2.4 – Вибірка часу виконання запиту

inner join FirstName (s)	,240	,236	,240	,239	,241	,242	,239	,245	,240	,242

Продовження таблиці 2.4 – Вибірка часу виконання запиту

inner join LastName (s)	,150	,147	,149	,147	,149	,150	,147	,151	,148	,149
Index Scan (s)	,068	,068	,069	,069	,070	,069	,070	,071	,069	,070
Total time (s)	,256	,530	,300	,548	,529	,539	,529	,529	,561	,547

Середній час оператора inner join FirstName: 0,240 с (стандартне відхилення: 0,002 с).

Середній час оператора inner join LastName: 0,149 с (стандартне відхилення: 0,001 с).

Середній час процесу Index Scan: 0,069 с (стандартне відхилення: 0,00 с).

Середній загальний час виконання: 2,487 с (стандартне відхилення: 0,105 с).

Проаналізувавши результати порівняння нормалізованої бд та ненормалізованої бд, можна зробити висновок, що час виконання представлених запитів майже однаковий, незважаючи на те, що у оптимізованому запиті додатково використовуються зв’язки з таблицями FirstNames та LastNames. Причиною цього, скоріш за все є велика робота над оптимізацією мови програмування до роботи з нормалізованими типами баз даних.

Вплив на продуктивність у нашому випадку був непомітним, але при цьому була зменшена вага головної таблиці і, що більш важливо, значно полегшився процес створення майбутніх запитів, пов’язаних з обраними даними.

Денормалізація для запитів, що інтенсивно читають.

Приклад завдання запиту: отримати ідентифікатори користувачів та покупок, які вони здійснили.

Для аналізу було створено таблиці Sales та DenormSales.

Sales, яка включає в себе ідентифікатор користувача, ідентифікатор умовного товару та час покупки, розмір: 100 000 рядків.

```
CREATE TABLE Sales (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    UserId INT NOT NULL,
    ProductId INT NOT NULL,
    Time DATETIME NOT NULL
);
SET NOCOUNT ON;

DECLARE @Counter INT = 1;

WHILE @Counter <= 100000
BEGIN
    INSERT INTO Sales (UserId, ProductId, Time)
    VALUES (
        FLOOR(RAND(CHECKSUM(NEWID())) * 500000) + 1, -- Random UserId from 1 to 500000
        FLOOR(RAND(CHECKSUM(NEWID())) * 1000) + 1,   -- Random ProductId from 1 to 1000
        DATEADD(SECOND, FLOOR(RAND(CHECKSUM(NEWID())) * 31536000), '2020-01-01') -- Random Time in 2020
    );

    SET @Counter = @Counter + 1;
END;
```

Рисунок 2.17 – Запит створення та заповнення нормалізованої таблиці

```
--Norm. query
SELECT u.[UserID]
      ,s.ProductId
  FROM [master].[dbo].[Users] u
    inner join sales s on u.UserId = s.userId
```

Рисунок 2.18 – Тестовий SELECT запит до нормалізованої БД

Вибірка результатів вимірювання запитів:

Таблиця 2.5 – Вибірка часу виконання запиту

inner join (s)	,116	,120	,116	,115	,119	,117	,120	,121	,115	,116
Index Scan Sales (s)	,007	,007	,007	,007	,007	,007	,007	,007	,007	,007

Продовження таблиці 2.5 – Вибірка часу виконання запиту

Index Scan Users (s)	,046	,049	,046	,046	,047	,046	,048	,047	,048	,048
Total time (s)	,321	,328	,325	,320	,330	,326	,327	,325	,322	,326

Середній час оператора inner join: 0,118 с (стандартне відхилення: 0,002 с).

Середній час оператора Index Scan Sales: 0,007 с (стандартне відхилення: 0,000 с, незначна дисперсія).

Середній час процесу Index Scan Users: 0,047 с (стандартне відхилення: 0,001 с).

Середній загальний час виконання: 0,325 с (стандартне відхилення: 0,003 с).

Процес денормалізації буде включати перенесення інформації зі стовпців таблиці sales до таблиці DenormSales, розмір таблиці залишився таким самим: 100 000 рядків.

```
CREATE TABLE DenormSales (
    Id INT IDENTITY(1,1) PRIMARY KEY,
    UserId INT NOT NULL,
    ProductId INT NOT NULL
);

INSERT INTO DenormSales (UserId, ProductId)
SELECT u.[UserID]
      ,s.ProductId
  FROM [master].[dbo].[Users] u
    inner join sales s on u.UserId = s.userId;
```

Рисунок 2.19 – Запит створення та заповнення ненормалізованої таблиці

```
--Denorm. query
SELECT [Id]
      ,[UserId]
      ,[ProductId]
  FROM [master].[dbo].[DenormSales]
```

Рисунок 2.20 – Тестовий SELECT запит до ненормалізованої БД

Вибірка результатів вимірювання запитів:

Таблиця 2.6 – Вибірка часу виконання запиту

Index Scan (s)	,015	,015	,015	,015	,015	,015	,015	,015	,015	,015
Total time (s)	,068	,072	,068	,065	,066	,067	,069	,067	,070	,66

Середній час оператора: 0,015 с (стандартне відхилення: 0,000 с, незначна дисперсія).

Середній загальний час виконання: 0,068 с (стандартне відхилення: 0,002 с).

Можна зробити висновок, що у випадках, коли зв’язок між таблицями є занадто ресурсовмістким, можна розглядати такий варіант реалізації, у нашому випадку, є більш ніж раціональним таке рішення, оскільки швидкість виконання помітно збільшилась.

Планування запитів

Планування запитів є одним з найважливіших методів оптимізації роботи бд, до яких мають доступ велика кількість користувачів водночас, завдяки їй можна без жодних проблем виконувати довгі та важкі для обчислювальних машин процеси, без впливу на роботу інших.

У нашому випадку, очевидно не має такої проблеми, тому докладно розглянути її неможливо, але можна розглянути можливий варіант виконання цього пункту, створення запиту, який буде виконуватись (створення резервної бд) та короткий огляд інструментів, які надають таку можливість.

Запит для резервної копії може мати подібний вигляд:

```
BACKUP DATABASE [master].[dbo].[Users]
TO DISK = 'C:\Backup\UsersTable.bak'
WITH FORMAT,
   NAME = 'Full Backup of Users table,
```

DESCRIPTION = 'Full backup taken on ' + CONVERT(VARCHAR, GETDATE());

Інструментарій, який можна використати в нашому випадку, це SQL Server Agent, вбудований в програмний додаток SSMS, саме він відповідальний за планування обраних запитів та завдань. Він надає достатньо широкий спектр налаштувань, головними перевагами якого є початок виконання при заданих значеннях максимального навантаження процесора, що є незамінним у випадках невідомого діапазону активного користування бд.

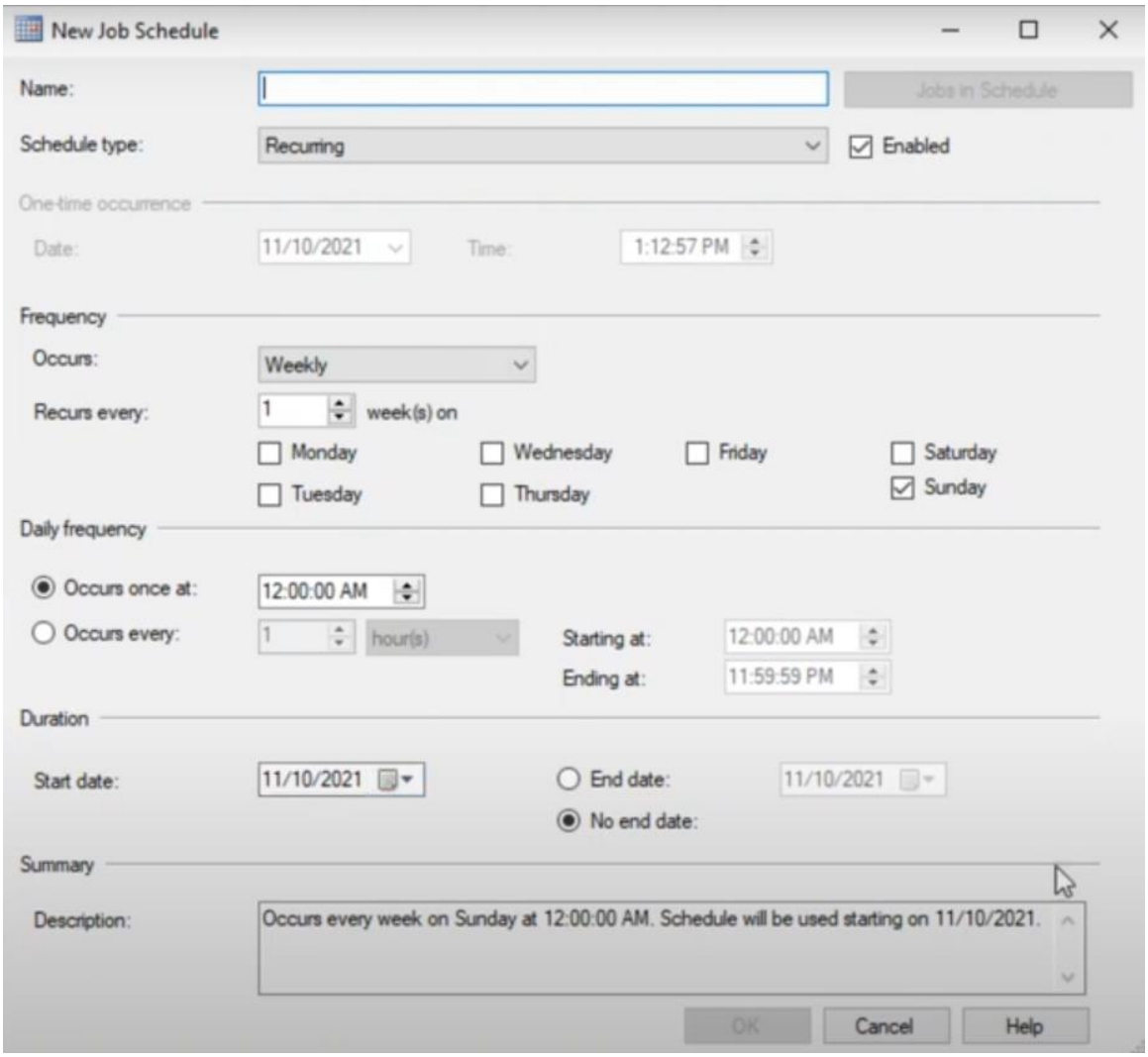


Рисунок 2.21 – Інтерфейс вбудованого інструментарію планування запитів

Профілювання та моніторинг запитів.

У теоретичній частині було приведено в приклад наявність процедури EXPLAIN, яка надає змогу проаналізувати виконання запиту, його вартість та деталі звернення до таблиць та їх індексів, ця функція має свої аналоги майже в кожній мові написання запитів, і MS SQL Server має один з найкращих аналітичних інструментів, які значно допомагають в оптимізації під час написанні запитів та виборах правильний методів виконання роботи.

Live Query Statistics дозволяє в реальному часі спостерігати за виконанням запиту, загальним деревом процесів, які відбуваються та скільки часу йде на кожен етап.

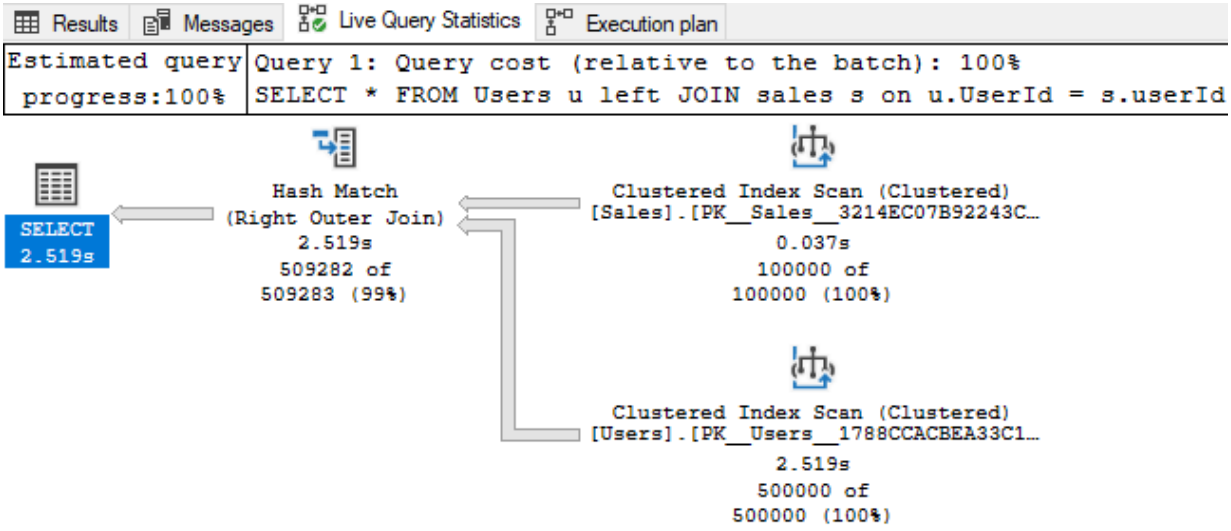


Рисунок 2.22 – Результуючий інтерфейс Live Query Statistics

Execution Plan дозволяє проаналізувати дерево процесів, зважаючи на вартість ЦП для кожного етапу, кількість пам’яті яка була використана та багато іншого, що буде дуже корисним для досвідчених користувачів, яким необхідно розробляти ресурсомісткі запити.

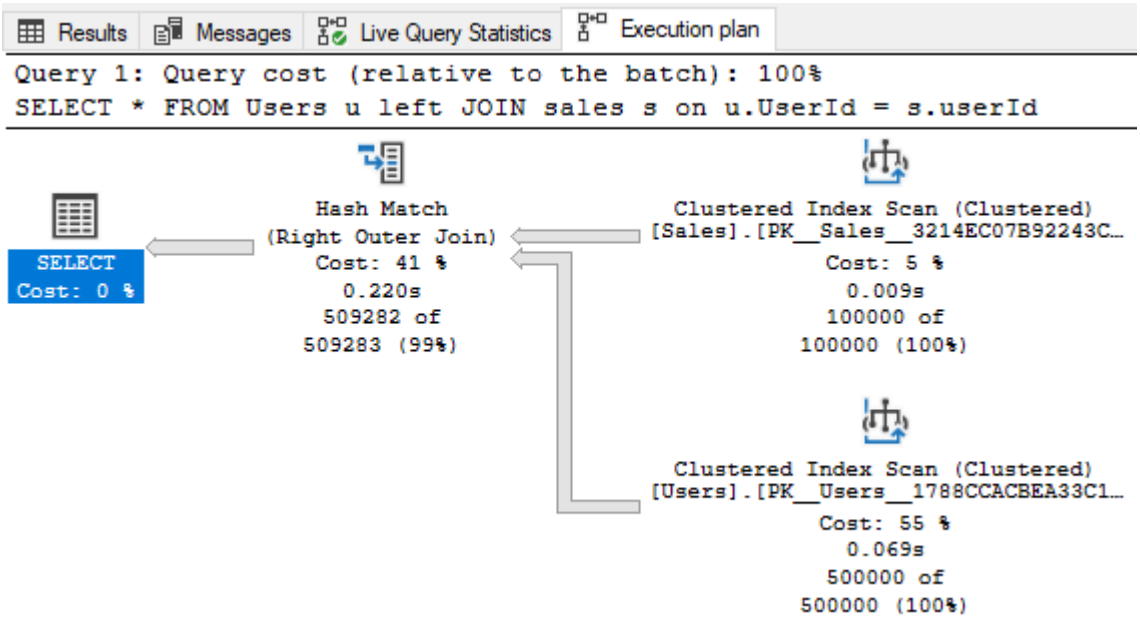


Рисунок 2.22 – Результуючий інтерфейс Execution Plan

Додатково, при наведенні на кожен з елементів плану, можна побачити багато додаткової інформації, як додатковий опис функцій, орієнтовна вартість виконання, кількість разів запуску та розрахований розмір результатів.

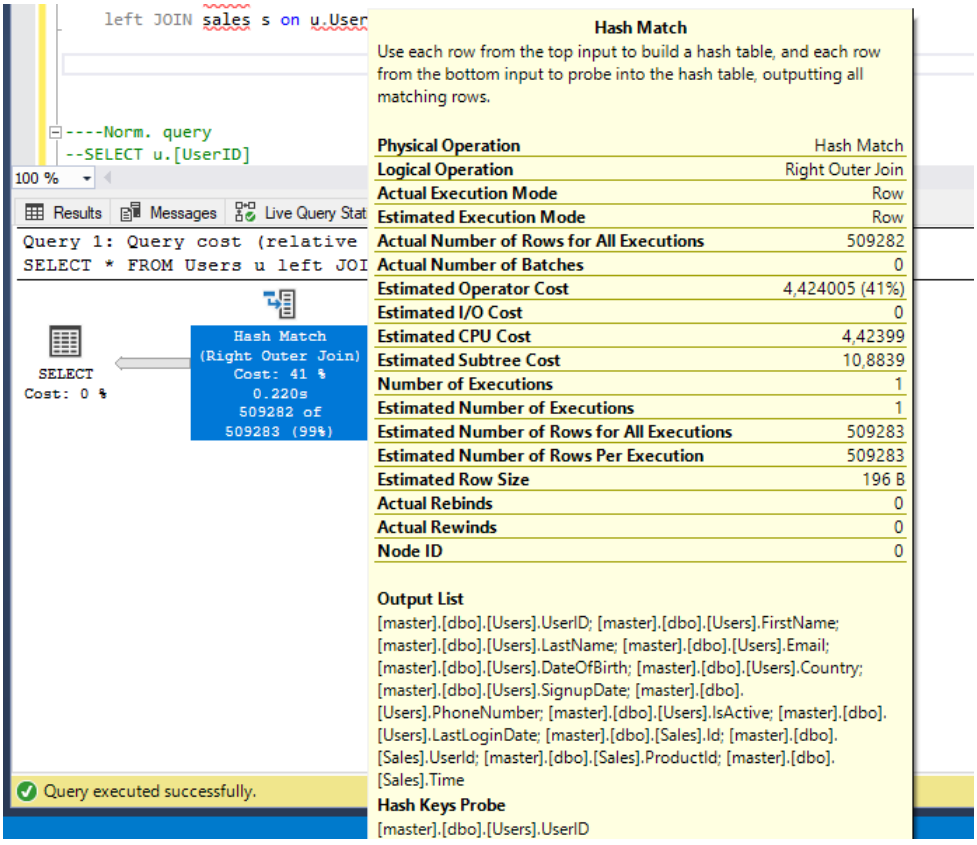


Рисунок 2.23 – Приклад додаткових даних одного з етапів виконання запитів в Execution Plan

Live Query Statistics та Execution Plan мають багато корисної інформації, яку складно зрозуміти без попереднього поглиблення у принципи роботи запитів, але навіть не маючи велику кількість досвіду можна побачити, що, наприклад, зв’язок обраних таблиць коштував 41% всієї вартості виконання, з чого можна зробити висновок та замислитись стосовно ненормалізації таблиць, які використовувались, або зміни підходу до завдання.

ВИСНОВКИ

У цій дипломній роботі розглянуто методи оптимізації запитів до баз даних із різними структурами та обсягами даних. Поставлені завдання включали аналіз ефективності запитів, дослідження підходів до їхнього вдосконалення, а також практичне впровадження цих методів. Було досліджено специфіку реляційних, NoSQL та ієрархічних баз даних, їхні переваги й недоліки в контексті продуктивності запитів.

Особлива увага приділялася практичній оптимізації запитів, включаючи використання індексів, нормалізацію та денормалізацію, застосування матеріалізованих представлень, кешування, профілювання та моніторингу запитів. За допомогою інструментів, таких як “Live Query Statistics” та “Execution Plan”, було оцінено вплив оптимізаційних методів на швидкість виконання запитів та витрати ресурсів.

Після аналізу результатів роботи можна було побачити, що кожен з представлених методів виявився ефективним та показував кращі результати в часі, у порівнянні з неоптимізованими альтернативами. З наданого великого списку можливих методів покращення запитів було практично продемонстровано такі способи, як оптимізація операторів SELECT, з покращенням виконання часу оператора в середньому з 0,142 с до 0,069 с, що є на 52% швидшим, нормалізація та денормалізація баз даних, що при правильному використанні не тільки дає можливість пришвидшити роботу (денормалізація даних привела до пришвидшення запиту в 4,7 разів), а й зменшити кількість простору, яке займають бази даних (нормалізація даних зменшила займаємий простір на 10%), додатково розглянутий метод планування запитів для виконання ресурсомістких запитів в сильно навантажених системах і останнє, що було продемонстровано це використання вбудованих інструментів Live Query Statistics та Execution Plan для більш глибокого аналізу продуктивності запитів.

ПЕРЕЛІК ПОСИЛАНЬ

1. MIIRO F. SQL and NoSQL databases [Електронний ресурс] / FABIAN MIIRO, MIKAEL NÄÄS. – Режим доступу: https://www.diva-portal.org/smash/get/diva2:927182/FULLTEXT01.pdf.
2. Date C.J. An Introduction to Database Systems [Електронний ресурс] / C.J. Date. – Режим доступу: https://lc.fie.umich.mx/~rodrigo/BD/An%20Introduction%20to%20Database%20Systems%208e%20By%20C%20J%20Date.pdf.
3. SQL Structured Query Language, Version 1.2 [Електронний ресурс] / Relational Systems Corporation. – Режим доступу: https://cdncontribute.geeksforgeeks.org/wp-content/uploads/SQL-Manual.pdf.
4. Sasaki B. M. Graph Databases For Beginners [Електронний ресурс] / B. M. Sasaki, J. Chao, R. Howard. – Режим доступу: https://neo4j.com/wp-content/themes/neo4jweb/assets/images/Graph_Databases_for_Beginners.pdf.
5. Суліма С. В. Метод оптимізації sql запитів системи управління базами даних [Електронний ресурс] / С. В. Суліма, О. Д. Єрмолаєв // Системи управління, навігації та зв'язку. – 2023. – No 2. – С. 151–157. – Режим доступу: https://www.researchgate.net/publication/372259794_METOD_OPTIMIZACII_SQL_ZAPITIV_SISTEMI_UPRAVLINNA_BAZAMI_DANIH.
6. Bolesta W. Analiza prędkoci wykonywania zapytań w wybranych bazach nie SQL-owych [Електронний ресурс] / W. Bolesta // Journal of Computer Sciences Institute – 2018. – V. 7. – P. 138–141. – Режим доступу: https://www.researchgate.net/publication/337173712_Analysis_of_query_execution_speed_in_the_selected_NoSQL_databases.
7. Fritchey G. SQL Server 2017 Query Performance Tuning [Електронний ресурс] / G. Fritchey. – 5-th edition. – New York : Springer, 2018. – 932 p. Режим доступу: https://dl.ebooksworld.ir/motoman/Apress.SQL.Server.2017.Query.Performance.Tuning.5th.Edition.www.EBooksWorld.ir.pdf.

8. Taipalus T. Database management system performance comparisons: A systematic literature review [Електронний ресурс] / T. Taipalus // Journal of Systems and Software. – 2024 – V. 208. – 111872. – Режим доступу: https://www.researchgate.net/publication/366846381_Database_management_system_performance_comparisons_A_systematic_literature_review

9. Hollingsworth M. Data Normalization, Denormalization, and the Forces of Darkness [Електронний ресурс] / M. Hollingsworth. – Режим доступу: https://slowanimals.com/melissa/WhitePapers/NormalizationDenormalizationWhitePaper.pdf

10. Moss J. M. Pro SQL Server Relational Database Design and Implementation : навч. посіб. [Текст] / J. M. Moss, L. Davidson. – Вид-во Apress; 5th ed. edition, 2016. – 828 с.

11. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics [Електронний ресурс] / K. Lee [et al] // SIGMOD ’16, [San Francisco], June 26–July. – San Francisco, 2016. – P. 1753–1764. – Режим доступу: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/qprog.pdf.