Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ. 170101.01.01.00.ПЗ

Виконав студент 2 курсу, група ІПЗм-21-1 _____ В. О. Бойко
Підпис, Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Ю. В. Форкун
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ І. В. Гурман
Підпис Ініціали, прізвище

До захисту допускаю:
завідувач кафедри інженерії програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

2 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
01.09.2022 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Бойку В'ячеславу Олександровичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Удосконалення методу матеріалізованих представлень у у реінжинірингу бази даних

Керівник проєкту (роботи) Форкун Юрій Вікторович, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.07.2022 р. № 83
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2022 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Дослідження предметної області та постановка задачі
2 Концепції, моделі та методи вирішення задачі
3 Технології вирішення задачі
4 Реалізація та тестування програмного засобу
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І. В., доцент	25.11.2022	2.12.2022
Нормоконтроль	Гумран І. В., доцент	29.11.2022	2.12.2022

7. Дата видачі завдання «01» вересня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження;	01.09-10.09.2022	
2 Робота над розділом 1 дипломної роботи – аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2022	
3 Робота над розділом 2 дипломної роботи – розробка моделей, методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2022	
4 Робота над науковими статтями	11.10-30.10.2022	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2022	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень; висновки до розділу	27.10-17.11.2022	
7 Попередній захист дипломної роботи	Листопад	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2022	
10 Підготовка до захисту дипломної роботи	з 01.12.2022 р	

Студент _____ Бойко В. О.
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ _____
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних».

Автор роботи: Бойко В’ячеслав Олександрович.

Керівник роботи: Форкун Юрій Вікторович.

Пояснювальна записка: 123 с., 25 рис., 1 табл., 3 дод., 31 джерело.

РЕІНЖИНІРИНГ, РЕДИЗАЙН, ДЕКОМПОЗИЦІЯ, БАЗА ДАНИХ, МАТЕРІАЛІЗОВАНЕ ПРЕДСТАВЛЕННЯ.

Об’єкт дослідження – реінжиніринг бази даних.

Мета дослідження – удосконалення методу матеріалізованих представлень, що дасть змогу використовувати усі переваги матеріалізації даних у комбінації із реалізованим на основі принципів кешування, алгоритмом синхронізації змін, а отже, підвищить показники продуктивності програмної системи та значно знизить вартість підтримки та супроводу апаратного та програмного забезпечення за рахунок досягнення ефективних результатів на етапі редизайну схеми та відмови від проведення декомпозиції сховища даних, як наступного етапу реінжинірингу.

У процесі дипломного проектування досліджено галузь реінжинірингу баз даних та сучасні методи і способи реінжинірингу, визначено невирішені проблеми у галузі, на основі чого удосконалено метод матеріалізованих представлень, виконано його програмну реалізацію у вигляді бібліотеки, навантажувальне тестування сховища даних та модульне тестування реалізованого плагіну. Нерозв’язані задачі запропоновано вирішити шляхом удосконалення методу матеріалізованих представлень, що полягає у використанні допоміжних таблиць для кешування вхідних змін та подальше їх застосування при синхронізації.

Удосконалений метод покращує продуктивність запитів вибірки даних, зокрема і тих, структура яких є не властивою для реляційної моделі.

Для програмної реалізації було використано платформу .NET, мову програмування C#, плагін Stopwatch для вимірювання швидкості методів та фреймворк XUnit для проведення юніт-тестування програмного засобу.

Проведені емпіричні випробування вдосконаленого підходу довели його придатність і ефективність у порівнянні з конкуруючими рішеннями у галузі реінжинірингу, а також життєздатність і функціональність програмного забезпечення, створеного з його використанням. Апробація отриманих результатів показала значний приріст продуктивності вибірки – більше, ніж на 80% у порівнянні із звичайною вибіркою. Отже, ІТ-підприємства, які бажають підвищити продуктивність свої програмних систем, можуть використовувати реалізований програмний засіб.

Підпис

01.12.2022
Дата

ABSTRACT

Master’s thesis: «Improving the method of materialized views in database reengineering».

Author: Boiko Viacheslav.

Head of research: Forkun Yurii.

Master’s thesis consists of: 123 p., 25 pc., 1 tb., 3 add., 31 srs.

REENGINEERING, REDESIGN, DECOMPOSITION, DATABASE, MATERIALIZED VIEW.

The subject of the research is the processes of database re-engineering.

The aim of the research is improvements of resiliency algorithm and development of new software on its basis which allows to handle and reduce the negative effect of failures (including partial failures).

The field of database reengineering, modern methods, and methods of reengineering were investigated during the study. Unsolved problems in the field were then identified, and the method of materialized representations was improved. Its software implementation took the form of a library, and load testing of the database and module testing of the implemented plugin were both carried out. Unsolved issues are suggested to be resolved by enhancing the materialized representations method, which involves using auxiliary tables to cache input changes and applying them again during synchronization process.

An improved method for the performance of data selection queries, the structure of which is not typical for the relational model. Due to the possibility of using data materialization in the form of representations on an ongoing basis, the developed method increases the performance of sample queries from tables with a complex or non-relational system structure of data organization.

The .NET platform, the C# programming language, the Stopwatch plugin for measuring the speed of methods, and the XUnit framework for unit testing of the software were used for software implementation.

Empirical testing of the improved approach has proven its suitability and effectiveness compared to competing reengineering solutions, as well as the viability and functionality of the software created using it. Approbation of the obtained results showed a significant increase in the productivity of the sample – more than 80% compared to the usual sample. Therefore, IT enterprises that want to improve the performance of their software systems can use the implemented software tool.

Signature

01.12.2022
Date

ЗМІСТ

ВСТУП

Ефективним інструментом пошуку інформації на сучасному етапі розвитку людства є Інтернет. З кожним роком інформації стає все більше, а застарілі програмні системи, що були розроблені десятиліття тому, не розраховані на велику кількість даних. А тому швидкість обробки даних є проблемою для більшості сучасних програмних систем. Побудова нової функціональності часто пов’язана зі зміною вимог бізнесу, і нова функціональність часто створюється із використанням застарілих фреймворків і шаблонів. Ключову роль у будь-яких мережевих програмних системах грає сховище даних, оскільки тип і структура організації інформації є найважливішими аспектами, що визначають швидкість обробки даних. А тому сховища даних є основними артефактами, що впливають на продуктивність програмного забезпечення. Сьогодні перед IT-фахівцями стоїть завдання постійного вдосконалення архітектури програмних систем для досягнення актуальних бізнес-цілей. В основі вдосконалення патернів лежать процеси реінжинірингу програмного забезпечення.

Реінжиніринг – це процес розробки нового програмного забезпечення або переосмислення та перепроектування вже існуючого для покращення його якісних атрибутів, підтримуваних ним функціональних можливостей, зниження вартості підтримки та ймовірності виникнення ризиків для замовника. Як складову глобального процесу, виокремлюють реінжиніринг даних (Data Re-engineering) – процес аналізу та зміни організації структури даних без зміни їх значень [1]. Процес реінжинірингу застосовується для реконструкції та реформування архітектури з метою підвищити показники продуктивності програмного забезпечення. Для підвищення показників продуктивності застосовують різні методи реінжинірингу сховища даних, більшість з яких хоч і вирішують проблеми продуктивності, проте підтримка та супровід таких систем є дороговартісним та складним процесом. А тому аналіз та удосконалення існуючих методів реінжинірингу бази даних є актуальним насьогодні.

Проаналізувавши наукові дослідження у сфері реінжинірингу бази даних, можна виділити наступні невирішені проблеми у галузі:

– при зберіганні даних різнорідної структури у сховищі одного типу виникає необхідність декомпозиції бази даних на декілька гетерогенних сховищ, що ускладнює процеси підтримки такої системи та підвищує витрати ресурсів на супровід апаратного та програмного забезпечення;

– практично неможливо оптимізувати складні запити у реляційних сховищах, що містять різнорідну структуру впорядкування даних, а реплікація даних лише ненадовго вирішить проблему;

– взаємодія між гетерогенними сховищами вимагає реалізації проміжного програмного забезпечення або використання наявних гібридних систем керування базами даних, що значно збільшить час та витрати на розробку ПЗ;

– застосування концепції матеріалізації даних значно прискорює виконання запитів вибірки, проте працюватиме лише для елементів програмної системи, які не потребують частого оновлення.

Таким чином, виникає потреба у вдосконаленні наявних методів реінжинірингу, які б підвищували продуктивність програмної системи, а витрати на підтримку та супровід були якомога меншими.

Тому завданням магістерської роботи є:

– визначення моделі та алгоритмів реінжинірингу бази даних, за допомогою яких можливе використання матеріалізації даних на постійній основі;

– проведення аналізу результатів та тестування роботи удосконаленого методу та визначення його ефективності у порівнянні із наявними.

Актуальність роботи полягає у тому, що основним впливовим чинником великої кількості програмних систем є сховища даних, оскільки характер та структура організації інформації має сильний вплив на продуктивність усієї програмної системи, а дослідження та удосконалення методу матеріалізованих представлень дасть змогу підвищити продуктивність ПЗ та зменшить витрати на супровід додаткових апаратних та програмних засобів.

Об’єкт дослідження – реінжиніринг бази даних.

Предмет дослідження – методи та способи реінжинірингу бази даних.

Мета дослідження – удосконалення методу матеріалізованих представлень, що дасть змогу використовувати усі переваги матеріалізації даних у комбінації із реалізованим на основі принципів кешування, алгоритмом синхронізації змін, а отже, підвищить показники продуктивності програмної системи та значно знизить вартість підтримки та супроводу апаратного та програмного забезпечення за рахунок досягнення ефективних результатів на етапі редизайну схеми та відмови від проведення декомпозиції сховища даних, як наступного етапу реінжинірингу.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

– провести теоретичний аналіз сфери реінжинірингу баз даних;

– виконати аналіз та порівняння методів реінжинірингу бази даних у програмних системах;

– описати наявні механізми реалізації реінжинірингу, проаналізувати їх переваги та недоліки;

– дослідити роботу матеріалізованих представлень та реалізовані на їх основі алгоритми, що підвищують продуктивність запитів вибірки;

– удосконалити метод матеріалізованих представлень у рамках реінжинірингу бази даних;

– провести тестування та практичну апробацію отриманих результатів, дослідити ефективність запропонованих рішень;

– проаналізувати отримані результати та сформувати рекомендації щодо доцільності впровадження результатів дослідження.

Для досягнення мети використано такі теоретичні методи дослідження:

– абстрагування – один з важливих методів, який дозволяє відкинути несуттєві параметри (від абстрагування напряму залежить ефективність моделі); аналіз та синтез – декомпозиція моделі на прості складові, виявлення зв’язків між компонентами і, відповідно, синтез цих структурних елементів у єдине ціле;

– формалізація – представлення моделі у вигляді програмного коду;

Також було застосовано такі емпіричні методи:

– спостереження (темою роботи є вдосконалення методу матеріалізованих представлень, але для того, щоб виділити корисні ознаки, які мають бути реалізовані у розроблюваних рішеннях, слід провести спостереження існуючих, визначити властивості та зв’язки між ними);

– експеримент (на етапі дослідження існуючих аналогів слід відтворити певні умови, які потрібні для аналізу реалізованих алгоритмів; пізніше цей же метод використовується для аналізу ефективності результативного алгоритму, який розроблено та імплементовано у ході роботи).

Наукова новизна отриманих результатів полягає у тому, що метод матеріалізованих представлень отримав подальший розвиток, що дозволило підвищити продуктивність запитів вибірки із бази даних та відмовитися від етапу декомпозиції сховища. Застосовані техніки кешування даних за допомогою допоміжних таблиць дозволили використання матеріалізації даних на постійній основі, а також оновлення матеріалізованих представлень лише у певний період часу, що значно підвищує продуктивність програмного забезпечення.

Практична цінність отриманих результатів полягає в успішному удосконаленні методу матеріалізованих представлень у рамках реінжинірингу бази даних. Завдяки поліпшеним характеристикам продуктивності у порівнянні з класичними рішеннями, удосконалений метод може бути успішно використаний при проектуванні гетерогенних систем, а програмне забезпечення, що буде його використовувати, матиме нижчу вартість підтримки та обслуговування і стабільно високу продуктивність.

Достовірність та обґрунтованість отриманих результатів підтверджується використанням у процесі перевірки удосконалених рішень експериментальними дослідженнями за допомогою відомих процедур проектування і тестування та наявністю наукової публікації у рецензованому виданні. За результатами дослідження опублікована стаття у фаховому науковому виданні.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, останніх досліджень та джерел

Реінжиніринг – це процес створення нового або перепроектування існуючого ПЗ з метою поліпшення характеристик якості, функціональності, що підтримується ним, зниження вартості супроводу та зменшення ймовірності виникнення значущих для замовника ризиків. При реінжинірингу використовується вже наявне в експлуатації у замовника програмного забезпечення, що виступає у ролі базового фундаменту для введення подальших оновлень.

Реінжиніринг він використовує бізнес-логіку, досвід, накопичений з часом, і функціональні можливості, які автоматизують звичайні бізнес-операції.

Реінжиніринг часто є складною справою, яка потребує висококваліфікованих розробників. Таке завдання недоступне програмістам низького та середнього рівня кваліфікації. Навіть досвідчені спеціалісти часто роблять помилки, оскільки реінжиніринг може охоплювати багато різних технологічних сфер та засобів. Як наслідок, для такого роду роботи потрібні люди з великим досвідом і знаннями багатьох технологій.

Тому реінжиніринг часто дорожчий, ніж розробка з нуля. Проте процес буде менш складним, якщо початкова архітектура програми була строгою та чіткою. Таким чином, реінжиніринг програмного продукту зазвичай визначається часом розробки нової архітектурної концепції та навичками спеціалістів [2].

Можна виділити декілька типів реінжинірингу, проте більш ваговим впливом на продуктивність програмної системи володіє реінжиніринг бази даних, оскільки сховище даних є основним чинником впливу на швидкість обробки інформації.

Існує декілька ситуацій, у яких може знадобитися реінжиніринг бази даних:

– проблеми з продуктивністю програмного забезпечення, викликані неефективною базою даних, тобто тою, у якій більшість методів по роботі з даними є неефективними та призводитимуть до деградації продуктивності програми;

– змінено вимоги до програмних продуктів, що вимагають оновлення структури даних;

– значні відмінності в останній версії бази даних порівняно з попередніми версіями, де ці відмінності вимагають змін у базі даних [3].

Фахівці виокремлюють декілька основних підходів реінжинірингу бази даних. Методи, що при цьому розглядаються, засновані на таких змінах бази даних:

– редизайн – якщо наявна структура даних має недоліки, її перепроектування може призвести до підвищення продуктивності;

– міграція або розширення поточної системи збереження даних базою такого самого типу. Використання іншої бази даних, яка має більш прийнятні показники, може підвищити продуктивність програмної системи або вирішити інші проблеми;

– перехід до бази даних іншого типу. У деяких випадках використання нереляційної бази даних замість реляційної може вирішити проблеми з продуктивністю. В інших випадках реляційна база даних може бути більш ефективною [3];

– використання баз даних різних типів одночасно. Деякі системи можуть містити різнорідні дані, а тому і методи зберігання інформації будуть різними. Такі системи, хоч і виправдовують зберігання даних у різнорідних базах, проте їх підтримка може виявитись не простим завданням, оскільки масштабування сховищ даних до нових типів вимагає значних витрат та ресурсів на підтримку життєздатності такої системи.

Усі ці техніки можна застосовувати одночасно. Деякі комп’ютерні програми використовують лише кілька прийомів. Для цього необхідно чітко розрахувати витрати на підтримку додаткових сховищ даних і серверів, детально вивчити план виконання поточних запитів до бази даних, виділити ті, що негативно впливають на продуктивність програми, і вибрати необхідні методи реінжинірингу на основі на дослідження та аналізу.

В залежності від методів, реінжиніринг розбивають на декілька стадій. Наприклад, при використанні декількох різнорідних сховищ даних, виокремлюють декілька кроків: створення нового сховища даних, декомпозиція таблиць у

реляційній БД, міграція частини даних, що потребують реорганізації структури, у нове сховище та створення шару взаємодії між ними, використання наявного зв'язувального програмного забезпечення для взаємодії між гетерогенними базами даних або ж введення у систему мультимодельних сховищ.

Одним із сучасних методів реінжинірингу програмних систем є використання мультимодельних баз даних. Різні сховища даних використовуються багатьма складними та сильно навантаженими системами для впорядкування інформації. Дані можна зберігати різними способами завдяки гнучкому підходу, який зменшує їх надмірність. Однак ефективність такої стратегії стає очевидною лише тоді, коли система має достатньо неоднорідно структурованих даних, щоб повністю виправдати їх зберігання в кількох типах сховищ. Але перехід до мульти-модельного сховища буде значною тратою часу та ресурсів, якщо лише кілька запитів впливають на зниження продуктивності програмної системи. При такому підході доступність різних технік моделювання обмежена та не підходить для відносно простих систем або проектів. При тому наявне програмне забезпечення, що підтримує мульти-модельну структуру (ArangoDB, OrientDB) не зріле, що означає виникнення додаткових ризиків у розробці програмних систем та сервісів, що описано у статті [4].

Бажано використовувати редизайн, коли схема реляційної бази даних містить таблиці, нормальна форма яких нижча за третю. Проте нормалізація не завжди приносить користь. Часом може бути вигідно зберігати деякі дані в денормалізованій формі, що є доцільним лише за певних обставин, тому важливо провести чітку межу для того, щоб визначитись чи застосовувати денормалізацію та чи принесе вона бажаних результатів. Наприклад, підрахунок кількості записів, які автоматично додаються під час створення запису, займає набагато більше часу, ніж підрахунок кількості записів у базі даних.

Оптимізація таблиці за допомогою ефективного використання індексу є складовою процесу редизайну. Одночасно оцінюються повільні запити та перевіряється наявність індексів у відповідних таблицях. Аналіз передбачає виконання реконструкції індексів, видалення невикористаних, зайвих і

застосування нових. Загальну продуктивність бази даних і програми можна підвищити за допомогою якісного редизайну шляхом реіндексації.

Проте індексація та оптимізація запитів не можуть вирішити проблему падіння продуктивності програмних системи при вибірці даних, які не пристосовані до зберігання у вигляді реляційних таблиць. Наприклад, у соціальних мережах дані можуть бути впорядковані у граф, при роботі з яким виконувати пошук глибинних зв'язків значно легше та швидше, аніж у реляційній моделі шляхом вкладених об'єднань таблиць або self-join-ів, що об'єднують зв'язки самі з собою. Хоча деякі дослідження і оптимізації проводились над такими запитами, що зазначено у [5], проте на великих наборах даних продуктивність все одно буде не високою, а навантаження на апаратну частину серверу БД – відчутним.

Тому постає завдання у розробці такого методу реінжинірингу, який би був відносно ефективним та не вимагав би декомпозицію реляційної бази разом з перенесенням даних у відокремлені предметно-орієнтовані сховища.

1.2 Аналіз існуючих методів та засобів реінжинірингу баз даних

Реінжиніринг застосовується до більшості програмних систем. Причини можуть бути різними, проте одна з найголовніших – зростання обсягів інформації. Протидіяти деградації продуктивності можна шляхом проведення оптимізації та реорганізації структуру програми, і сховище даних – не виняток. Таким чином, для аналізу та дослідження було взято ряд програмних систем.

Соціальна мережа Instagram – сервіс для поширення фотографій і пошуку публікацій користувачів. Ним користується значна кількість населення, а тому причинами реінжинірингу архітектури, у тому числі і сховища даних, є стрімке збільшення обсягів даних. Щодня сотні тисяч користувачів одночасно публікують медіаконтент за допомогою сервісу. Тому стандартний класичний підхід до збереження даних – використання однієї реляційної СКБД, що у Instagram

використовувалася на початку його заснування, не працює, оскільки через велику кількість інформації виникають затримки та продуктивність стрімко знижується.

Як вказано у статті [6], у процесі реінжинірингу реляційної бази даних було виконано її декомпозицію на допоміжні сховища, що дозволяють ефективно виконувати запити до сервісу.

Instagram в основному використовує дві серверні системи баз даних: PostgreSQL і Cassandra. Обидва сховища мають розвинену структуру реплікації та добре працюють як глобально узгоджена система сховищ.

Дані, які зберігаються на серверах, зіставляються з глобальними даними. Враховуючи потенційну затримку, мета полягає у тому, щоб зрештою досягти рівномірності в усіх центрах обробки. Наявність репліки читання в кожному регіоні запобігає перехресному читанню з веб-серверів, оскільки операції читання є значно більш поширеними, ніж операції запису.

Як зазначено у [7], оскільки Cassandra має можливості до швидкого запису данних, Instagram використовує її як загальну службу зберігання ключів і значень, щоб підтримувати стрічку фотографій користувачів, обробку повідомлень, а також для виявлення шахрайства.

Отже, реінжиніринг шляхом декомпозиції бази даних та використання окремих сховищ для конкретних цілей є вдалим рішенням для Інстаграму. Проте, недоліком такого рішення є значні витрати на підтримку серверів із базами даних різних типів. Створення кворуму вузлів у такій БД, як Cassandra, для обробки відмов та підтримки консистентності вимагає більше ресурсів для супроводу.

Крім того, як показано в [8], Instagram зіткнувся з великою кількістю реплік під час глобального розширення, оскільки для кожного екземпляра бази даних потрібно було створити хоча б одну. Крім того, на запити щодо кворуму мали відповідати дві копії, розташовані близько одна до одної. Коли запити надходять з інших континентів, три дублікати, необхідні для виконання запиту, розташовувались географічно далеко один від одного. Продуктивність постраждала через таке погане налаштування. Щоб вирішити проблему, Instagram

прийняв рішення припинити дублювання даних і використовувати лише сховище, розташоване поблизу місця, де вони були згенеровані.

Також, у Instagram було проведено редизайн лічильників, що рахують кількість вподобань. Замість запиту «select count(*)», як зазначається у [6], щоб зменшити ресурси, було денормалізовано лічильник вподобань для публікації. Щоразу, коли надходить новий відповідний запит, значення лічильнику у базі даних збільшується. Таким чином, кожне зчитування підрахунку вибирається за допомогою простого «select», який є набагато ефективнішим, оскільки при підрахунку кількості більшість систем керування базами даних вдаються до блокування таблиць, оскільки консистентність при підрахунку повинна зберігатись, а кількість даних на момент проведення повинна бути незмінною.

Так як Postgres має можливість зберігати і масиви у таблиці та представляє набір функцій для роботи з ними, Instagram зберігає певні текстові дані, що не вимагають детального повнотекстового пошуку у вигляді масиву. Важливо підкреслити, що оскільки PostgreSQL використовує обмежений розмір сторінки (часто 8 КБ) і не дозволяє кортежам охоплювати кілька сторінок, ефективна робота з масивом можлива лише за його невеликого розміру. Тому неможливо безпосередньо зберігати дуже великі значення полів. Щоб подолати це обмеження, великі значення полів стискаються та (або) розбиваються на кілька фізичних рядків. Техніка відома як TOAST [9]. Її рекомендовано уникати, оскільки вона хоч і застосовується для підвищення продуктивності при роботі з великим кортежами, проте продуктивність на необмежено великих наборах даних буде деградувати. Інстаграм же використовує таке збереження, оскільки гарантовано розмір масиву не перевищуватиме встановлений.

Отже, можна виділити основні методи реінжинірингу, які застосовувались у соціальній мережі Instagram:

– денормалізація лічильників записів – винесення їх у окремі поля та використання прямого запиту при читанні;

– збереження невеликої кількості даних дозволяє відмовитись від додаткової таблиці та використовувати масиви;

– декомпозиція бази даних, та винесення деяких відношень у окреме колонко-орієнтоване нереляційне сховище Cassandra, що позитивно впливає на швидкість запису даних.

Також було проведено аналіз реінжинірингу програмної системи Hotel Service Optimization компанії Amadeus, що відповідає за керування роботою персоналу у готелях та готельних корпораціях. Як і більшість програмних систем на ранніх стадіях, дана програма містила лише одне реляційне сховище даних – MS SQL Server. При декомпозиції на мікросервіси певний час база даних залишалась спільною, що викликало проблеми з продуктивністю. Також було визначено, що повнотекстовий пошук працює повільно, оскільки MS SQL Server не підтримує ефективний парсинг тексту для пошуку. Тому, допоміжне сховище було обрано документо-орієнтованого типу. Однією з відомих баз даних для повнотекстового пошуку є Elastic Search.

Сам процес реінжинірингу проведено у декілька етапів. Першим із них є аналіз потенційного приросту продуктивності та доцільність переходу. Для цього аналізується нове сховище даних, завантажуються тестові дані та виконується навантажувальне тестування. При цьому після завершення результати порівнюються і приймається відповідне рішення.

Наступним етапом є декомпозиція реляційної бази даних із вилученням сутностей, над якими повинен виконуватись повнотекстовий пошук.

Далі необхідно вибрати засіб для комбінування та взаємодії двох сховищ з метою синхронізації змін та вибірки шляхом об’єднання даних.

У статті [10] описано використання гібридного підходу до зберігання даних, принцип роботи системи з використанням гетерогенних баз даних, подано схему архітектури зв’язувального програмного забезпечення між гетерогенними сховищами, детальний опис компонентів архітектури, таких як шар парсингу синтаксису та обчислювального ядра. Мову для доступу до даних, що містяться у різнорідних сховищах, обрано SQL. Крім того, проведено тестування системи, виконано основні тестові запити та показано результат їх виконання.

Використовувалися такі типи запитів: проста вибірка з одної з баз даних, вибірка з об’єднанням різних сховищ, агрегацію та вкладені запити.

Подібний підхід застосовано і у даній системі. У якості обгортки виступає шар зі спільним SQL-інтерфейсом для виконання запитів вибірки із різних сховищ із підтримкою об’єднання контекстів. Також реалізовано синхронізацію даних при записі, оновленні та видаленні. Для цього було підключено Logstash – систему збору даних з відкритим кодом, що може отримувати дані з різних типів сховищ, перетворювати дані та поміщати у сховище. Спочатку він використовувався для збору журналів, але його можливості виходять за рамки такого способу використання. Logstash має багато плагінів, які дозволяють читати дані з певних джерел. Таким чином дана система збору даних представляє можливості синхронізації змін, що дозволяє у деяких випадках надсилати запити на читання та пошук інформації лише у ElasticSearch, а усі інші – у реляційне сховище, після чого відбувається синхронізація, а отже, зберігається цілісність. Усі помилки записуються у логи. Коли дані переміщуються від джерела до сховища, фільтри Logstash аналізують кожну подію, ідентифікують іменовані поля для побудови структури та перетворюють їх у загальний формат для більш потужного аналізу та комерційної цінності. Система має здатність взаємодіяти із іншими базами даних та встановлювати зв’язки із метою забезпечення багатомодельної архівації метричних даних програмної системи [11]. Спрощену схему гетерогенного підходу до впорядкування різнорідних даних подано на рисунку 1.

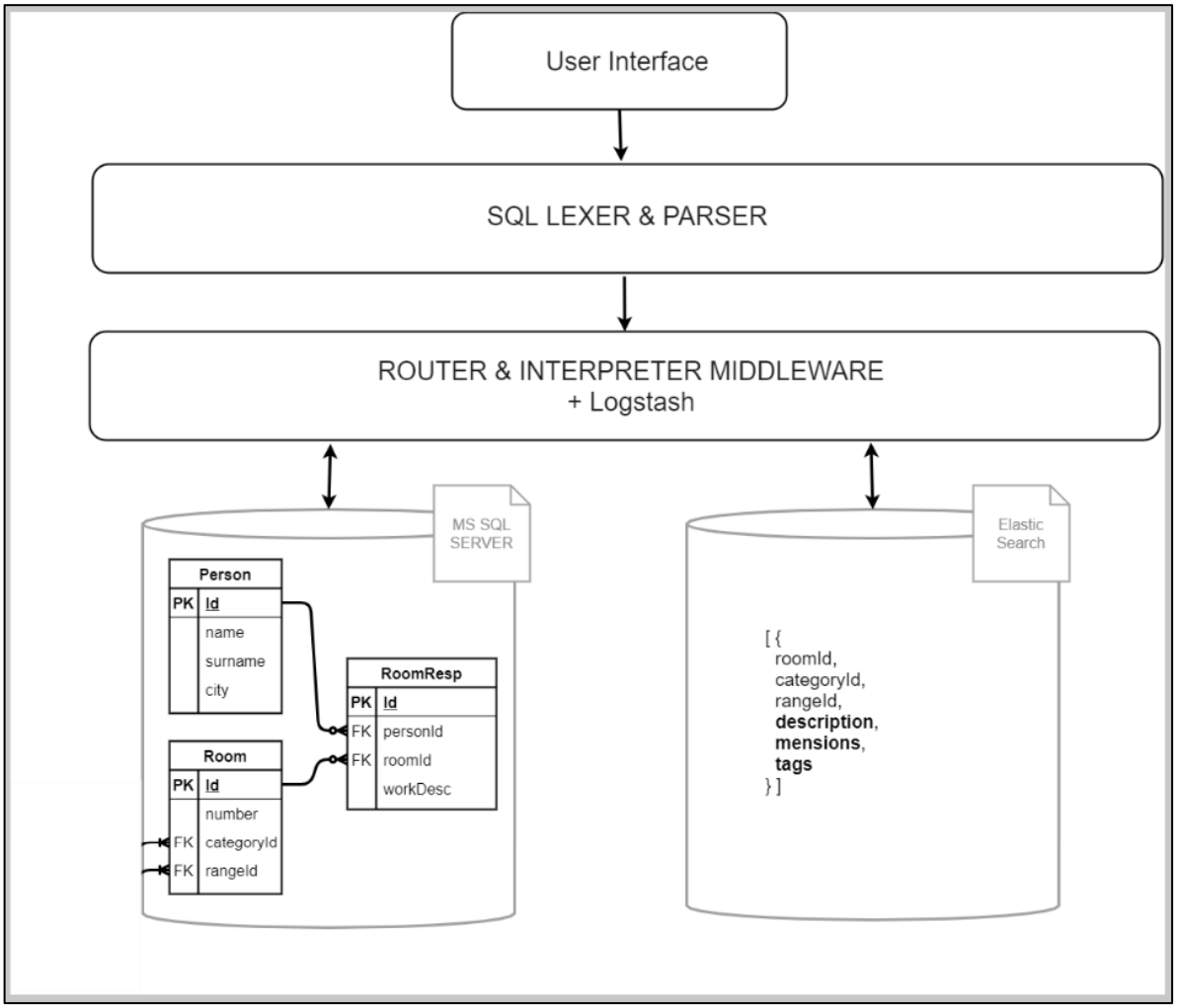


Рисунок 1.1 – Спрощена схема гетерогенного підходу до впорядкування та керування різнорідними даними

Як видно з рисунку 1.1, інформація про кімнати готелю, що потребує повнотекстового поля (поля description, mentions та tags) переміщена у нереляційне сховище ElasticSearch. При цьому з метою збереження зв’язків із таблицями категорії та рангу кімнати, поля, що відповідають за зовнішні ключі – categoryId та rangeId було продубльовано у ElasticSearch. Таким чином цілісність даних зберігається, а запити спочатку посилаються на шар лексера та парсера з метою трансформації SQL-запиту у нативні запити до гетерогенних сховищ.

Таким чином, у результаті проведеного реінжинірингу показники продуктивності текстового пошуку у системі оптимізації готельного бізнесу зросли, а навантаження було розподілено рівномірно.

Міграція даних до багатомодельних сховищ — це інший підхід до реінжинірингу. Якщо недостатньо ресурсів для підтримки всіх різнорідних сховищ системи, можна застосувати цю стратегію. У статті [12] вказано, що, незважаючи на те, що мультимодельні бази даних ще у процесі розвитку, за останні роки вони зарекомендували себе на ринку як універсальні рішення, що розв’язують основні проблеми, а саме, зберігають дані різнорідного типу та пропонують уніфікований доступ до них. У статті також порівняно основні типи таких СКБД. Тому сучасним методом реінжинірингу сховища даних можна вважати використання мультимодельних баз, але, по-суті, це уже знайомий метод міграції, що застосовується на іншому рівні.

Деякі дослідження проводилися і у темі використання матеріалізованих представлень у рамках реінжинірингу. Так, у статті [13] описано використання, їх у комбінації з централізованим сховищем даним Data Warehouse. Там міститься інформація, зібрану протягом тривалого часу з багатьох джерел даних. Запити до такого сховища є складними за своєю природою та займають досить тривалий проміжок часу при виконанні. Час відповіді можна скоротити шляхом створення матеріалізованих представлень і зберігання їх у сховищі даних.

Проте недоліком матеріалізованих представлень є їх статичність, тобто, сама їх концепція полягає у тому, щоб зберігати великі набори даних уже у тому вигляді, у якому необхідно їх отримати. Це свого роду кешування результатів запиту і для оновлення представлень і синхронізації даних існують певні механізми. У деяких СКБД матеріалізовані представлення оновлюються автоматично (OracleDb), у інших – лише вручну (PostgreSQL). Також необхідно пам’ятати, що при синхронізації представлень та оригінальної таблиці, виконується запит із вибіркою усіх даних з оригінальних таблиць, що, якщо виконувати часто, знизить продуктивність системи, в залежності від складності запиту.

Саме тому їх зазвичай використовують для отримання даних із таблиць, що рідко оновлюються, а, отже, потреба у синхронізації виникає лише один або декілька разів у певний проміжок часу [14].

У програмній системі у рамках редизайну схеми даних було впроваджено матеріалізовані представлення. На одну таблицю може бути створено декілька матеріалізованих представлень в залежності від кількості підготовлених запитів. Представлення застосовувались для тестування ефективності обробки запитів на архівних базах даних, що не потребують постійного оновлення. Вибірка даних при простому запиті у будь-якому разі складає O(log(n)) при застосуванні первинного ключа, а, отже і індексу.

Так як дані не потребують швидкого оновлення, оскільки це архівна інформація, то для автоматичного оновлення представлення було написано заплановану процедуру, що проводить оновлення один раз на день.

Представлення мають досить сильний потенціал у використанні при роботі і з даними, що підлягають частому оновленню, а недолік оновлення можна усунути. У цьому і полягає мета для наступних досліджень – у розробки методу редизайну структури бази даних у рамках реінжинірингу сховища даних програмних систем.

1.3 Методологічні підходи до вирішення задачі удосконалення реінжинірингу бази даних

Процес реінжинірингу бази даних – це запланована діяльність, яка складається з кількох етапів і має на меті покращити показники продуктивності програмної системи. При тому деякі етапи можуть і не знадобитись у випадку досягнення очікуваних результатів на одному із них. На кожній стадії до та після проведення прийнято проводити навантажувальне тестування для порівняння результатів та визначення приросту продуктивності. Крім того, перед кожним етапом проводиться експертиза доцільності його впровадження. Аналізуються уже наявні рішення конкретних проблем, що можуть з’явитись у результаті планування.

На етапі редизайну потрібно провести аналіз схеми, відповідність відношень нормальним формам та вияснити, чи впливає надмірна або недостатня нормалізація на загальну продуктивність програмної системи. Якщо є негативний вплив, то

необхідно його усунути шляхом реорганізації табличної структури. Вкінці необхідно виконати навантажувальне тестування.

Перевірка запитів також важливий елемент у проведенні редизайну. Більшість баз даних мають вбудовані процедури для відстеження неефективних запитів і, якщо їх викликати, можна одразу віднайти ті, що негативно впливають на продуктивність. Для виправлення проблеми такі запити необхідно переписати на більш оптимальні, а час виконання можна порівняти, використовуючи дані із плану виконання запитів. Особливо необхідно звернути увагу на автоматично згенеровані запити. Вони виникають унаслідок використання ORM-системи у програмі. Такі запити необхідно теж переписувати вручну.

Службові операції, такі, як фрагментації індексів, очищення пустих кортежів тощо, повинні виконуватись незалежно від того, чи проводиться загальний редизайн, чи ні. На основі перевірок продуктивності можна встановити доцільність проведення таких дій, але у рамках редизайну необхідно їх виконати обов’язково.

Фрагментація індексів є службовою операцією, але може бути проведено у рамках редизайну, оскільки індекси мають великий вплив на продуктивність. Наприклад, у MS SQL Server рекомендовано виправляти проблеми фрагментації індексу шляхом їх перебудови, якщо відсоток фрагментації перевищує 30%, при цьому рекомендовано вирішити проблему фрагментації індексу шляхом реорганізації індексу, якщо відсоток фрагментації перевищує 5% і менше 30%. Операція реорганізації індексу використовує лише один потік. Для визначення ступеня фрагментації у більшості СКБД є спеціальні вбудовані процедури. Крім того можна проаналізувати використання тих чи інших індексів та їх застосування у конкретних таблицях, видалити зайві та додати необхідні.

У деяких СКБД кортежі, що видалені, фізично не видаляються із таблиці; вони залишаються присутніми, доки не буде виконано спеціальну процедуру очищення. У PostgreSQL, наприклад, така процедура називається Vacuum, та застосовується для зменшення розмірів таблиці шляхом видалення пустих кортежів. Тому таку процедуру необхідно виконувати часто, особливо у часто оновлювальних таблицях. Зазвичай для цього створюють заплановану процедуру,

яка буде моніторити стан часто оновлювальних таблиць, та, у разі виникнення критичної кількості пустих кортежів – їх видаляти.

Після завершення проведення редизайну виконують навантажувальне тестування самої бази даних та програмної системи в цілому. Існує велика кількість засобів для проведення такого тестування. Наприклад, у PostgreSQL для цього використовується утиліта pgbench – це проста програма для виконання тестів на PostgreSQL. Він виконує ту саму послідовність команд SQL знову і знову, можливо, у кількох одночасних сесіях бази даних, а потім обчислює середню швидкість транзакцій (транзакцій за секунду). За замовчуванням pgbench тестує сценарій приблизно на основі TPC-B та п’ять команд SELECT, UPDATE і INSERT на транзакцію. Є можливість перевірити інші випадки, написавши власні файли сценаріїв транзакцій [15].

Для тестування продуктивності усього додатку одним із популярних рішень є програмне забезпечення Apache JMeter. Він призначений для навантажувального тестування функціональної поведінки та вимірювання продуктивності. Використовують JMeter для аналізу та вимірювання продуктивності веб-додатків сервісів. Тестування продуктивності означає перевірку веб-програми на велике навантаження, багаторазовий і одночасний трафік користувачів. Може використовуватись як для тестування цілого додатку, так і лише сховищ даних [16].

Після тестування порівнюються результати та встановлюється приріст продуктивності. І на основі нього приймається рішення про те, чи необхідно проводити наступні етапи реінжинірингу. Якщо рівень показника приросту продуктивності задовільний, або рівний спланованому, то процес реінжинірингу завершується на першому етапі. Якщо – ні, аналізується подальша діяльність відповідно до наступних етапів реінжинірингу.

Якщо є вимоги до збереження даних, що є неефективними у реляційній структурі і вимагають інших методів обробки та збереження, тоді є два варіанти проведення подальшого реінжинірингу: міграція даних у сховище іншого типу та комбінування декількох сховищ даних. У першому випадку повне переосмислення структури організації інформації. Перед проведенням такого роду реінжинірингу

необхідно зважити усі переваги та недоліки шляхом аналізу потенціального приросту продуктивності. Виконувати міграцію необхідно зі збереженням старої версії сховища даних, використавши його як резервне у разі негативних наслідків.

У публікації [17] наведено приклад методу реінжинірингу шляхом міграції даних з реляційного сховища до графової бази даних. Спочатку створюються правила конвертації, потім згідно правил виконується міграція. Також можуть бути задіяні специфічні програмні засоби для оптимізації міграції. У наведеній статті – це Neo4J Parallel Batch Importer API, що дозволяє імпорутвати дані згідно правил трансформації даних. Також після міграції схеми та даних необхідно виконати реінжиніринг запитів. Для цього можна використати готові сервіси або проекти, що знаходяться у відкритому доступі, наприклад, sql2cypher. Після завершення необхідно провести тестування продуктивності.

Інший випадок, коли необхідно зберігати лише частину даних у окремій, пристосованій до цього, базі даних. Для цього треба визначити, чи бази даних повинні мати зв’язок, чи вони будуть повністю ізольованими. У першому випадку потрібно проаналізувати наявні рішення, що дозволяють автоматизувати процес обробки даних декількома сховищами. Це може бути створене з нуля зв’язуюче програмне забезпечення, як описано у [10], що синхронізує декілька гетерогенних сховищ та дозволяє виконувати вибірку одразу із декількох баз. Якщо додаток вимагає більш оптимального механізму взаємодії між декількома СКБД, необхідно розглянути можливість використання мультимодельних баз даних, таких як, наприклад, ArangoDB або OrientDB. Процес міграції у такі сховища є аналогічним до перенсення даних у інші бази даних.

Додатково треба врахувати можливості мультимодельних баз даних, провести тестування навантаженням для з’ясування їх реальних характеристик, потужностей та доцільність використання при реінжинірингу.

Після завершення необхідно провести тестування продуктивності та проаналізувати показники приросту. Якщо вони не змінились або залишились на рівні якогось з етапів реінжинірингу, необхідно повернутись до цього етапу, оскільки нема сенсу впроваджувати додаткові системи керування базами даних,

якщо це ніяк не впливає на продуктивність. Нові рішення, що були впроваджені на новій стадії зберегти, як шаблон для подальших досліджень та вдосконалень. Пізніше їх можна буде використати, якщо вимоги до програмного забезпечення зміняться і потребуватимуть переосмислення структури певних елементів бази.

Підсумовуючи, можна відмітити, що основними недоліками другого етапу реінжинірингу є витрата більшої кількості ресурсів, складність тестування та підтримки таких систем. Тому виникає необхідність у розробці методів, що забезпечать якісний реінжиніринг на етапі редизайну з урахуванням різних типів структури організації інформації.

1.4 Висновки. Постановка задачі

У зв’язку з постійним ростом об’ємів даних, насьогодні виникає необхідність в удосконаленні наявних методів реінжинірингу сховищ, оскільки продуктивність системи напряму залежить від здатності серверів баз даних обробляти великі об’єми даних. Особливо важлива швидкість читання, тому що це є однією з основних операцій, що у своїй підмножині об’єднує фільтрування та сортування. Такі операції на великих наборах даних можуть виконуватись дуже довго, особливо, якщо структура даних є різнорідною. Як наслідок – велика ймовірність затримки у роботі такої системи, а тому процес реінжинірингу бази даних може покращити ситуацію. Існують різні рішення, що можуть покращити продуктивність, проте більшість із них є дороговартісними за рахунок міграції на інші сервіси або використання додаткового програмного забезпечення.

Тому, проаналізувавши номенклатуру об’єктів у рамках лише одного сховища, було взято до уваги концепції матеріалізації даних, що у БД реалізують матеріалізовані представлення. Встановлено, що недоліком є неможливість забезпечення частого оновлення у багатьох СКБД, а отже, матеріалізовані представлення можуть зберігати лише дані, що оновлюються рідко. Крім того, запити вибірки даних невластивої структури є повільними, навіть за використання

оптимізації, а тому виникає необхідність у застосуванні їх допоміжного кешування, що підвищить продуктивність запитів вибірки та подальші етапи та методи реінжинірингу сховищ будуть не потрібні.

У зв’язку з цим усі дослідження у роботі будуть спрямовані на удосконалення матеріалізації та введення матеріалізованих представлень у постійне використання за рахунок розробки спеціальних алгоритмів, що ґрунтуватимуться на відомих принципах.

Наступним етапом роботи є аналіз концепцій, моделей та методів вирішення поставленої задачі, визначення та детальний аналіз етапів, моделей та алгоритмів проведення процесу реінжинірингу бази даних.

2 КОНЦЕПЦІЇ, МОДЕЛІ ТА МЕТОДИ ВИРІШЕННЯ ЗАДАЧІ

2.1 Концепції та етапи процесу реінжинірингу бази даних

Традиційні методи реінжинірингу сховища даних програмного забезпечення є досить ефективними та здатними підвищувати продуктивність програмної системи при вдалому застосуванні їх комбінацій. Методи реінжинірингу можна розділити на два ключових етапи: методи, що дозволяють виконати оптимізацію у повному обсязі, не покидаючи рамки бази даних конкретного типу – редизайн, та ті, при яких необхідно використовувати інші сховища даних – декомпозиція та міграція. Тому розглянемо основні дієві методи реінжинірингу бази даних. Методи було вибрано з урахуванням реляційного типу сховища, оскільки воно є найпопулярнішим у корпоративних програмних системах.

Одним із загальновідомих методів реінжинірингу на етапі редизайну є оптимізація запитів. Рано чи пізно продуктивність у робочій бази даних стає проблемою. Запити, що потребують багато часу на виконання, не лише погіршують продуктивність серверів і програм, споживаючи значні системні ресурси, але також можуть призвести до блокування таблиці та пошкодження даних. Тому оптимізація запитів стає необхідною, щоб запобігти зниженню продуктивності. Вона насамперед означає вибір, а потім послідовність у певному порядку різних положень SQL, або іншої мови для формулювання ефективного запиту з кількох планів запитів шляхом їх порівняння на основі вартості залучених ресурсів і часу відповіді. Метою оптимізації запитів є забезпечення мінімального часу відповіді та максимальної пропускної здатності, тобто ефективного використання ресурсів [18].

Щоб покращити продуктивність запитів, розробникам і адміністраторам баз даних необхідно розуміти оптимізатор і методи, які він використовує для вибору шляху доступу та підготовки плану виконання запиту. Налаштування запитів передбачає знання таких методів, як оптимізатори на основі вартості та евристики, а також аналізатори, що генерують так званий план запиту – список усіх операцій, які відбувалися у рушії бузи даних під час виконання запити та його час.

Найкращий спосіб налаштувати продуктивність – спробувати написати запити кількома різними способами та порівняти їхні плани читання та виконання. У статті [19] запропоновано техніки, які можна використовувати для оптимізації. Оптимізація запитів має дуже великий вплив на СУБД, вона здатна значно покращити продуктивність. Проте, не завжди оптимізація запитів може допомогти вирішити усі проблеми з продуктивністю.

Використання індексів (індексування) – ще один важливий метод редизайну бази даних. Раціональне використання технології індексування має важливе значення для покращення продуктивності запитів до бази даних. Більшість реляційних баз даних використовують структуру B^+ дерево для зберігання індексів. Кластерний індекс може змінювати фізичне розташування даних. І кластерний індекс, і некластерний залежать від B^+ індексного дерева при запиті даних. Некластерний індекс повинен залежати або від вказівника рядка даних, або від ключа кластерного індексу для пошуку. Замість того, щоб сканувати всю таблицю для отримання результатів, можна зменшити кількість операцій вводу-виводу або вибірки сторінок за допомогою структур індексів, таких як B-Tree або хеш-індекси, щоб отримати дані швидше. Найзручніший спосіб розглядати індекс у вигляді словника даних, що має ключ та значення, яке можна отримати по ключу. Коли змінюється запис і відповідне значення індексованого стовпця в кластерному індексі, базі даних може знадобитися перемістити весь рядок в окрему нову позицію, щоб зберегти рядки в порядку сортування. Ця дія фактично перетворюється на запит видалення DELETE, за яким слідує INSERT, і це знижує його продуктивність. Кластерний індекс у таблиці часто може бути доступним у стовпці первинного ключа або зовнішнього ключа, оскільки значення ключа зазвичай не змінюються після введення запису в базу даних. Варто зазначити, що індексування корисне у багатьох випадках лише при запитах вибірки, оскільки структура даних, що застосовується при індексації пристосована до швидкого читання, проте повільнішого оновлення та видалення даних [20].

Якщо ж після аналізу виявилось, що індексів достатньо, необхідно перевірити їх стан із метою виявити ті, які необхідно оновити. Перебудова індексів

також є одним із основних методів редизайну бази даних. У публікації [21] запропоновано вирішення проблеми автономної роботи із фрагментованими індексами в реляційній СКБД шляхом відстеження поточного стану індексів і, якщо вони стають фрагментованими, автоматично перебудовувати їх. Перевагами методу є те, що заходи із перебудови індексів, які проводяться на постійній основі дозволяють прискорити операції у базах даних.

Оптимізатор запитів в багатьох реляційних базах даних у більшості випадків вибирає високоефективний індекс. Загальна стратегія розробки індексів надає оптимізатору запитів різноманітні варіанти, щоб визначити найкращий. Цей процес зменшує загальний час аналізу та показує хорошу ефективність у різноманітних ситуаціях. Щоб визначити, які індекси використовує оптимізатор для певного запиту, можна використати для аналізу план виконання запиту. Також не варто завжди ототожнювати використання індексу з хорошою продуктивністю. Іноді також можливо, що неправильний вибір індексу також може призвести до зниження продуктивності. Таким чином, робота оптимізатора запитів полягає у виборі індексу або групування індексів лише тоді, коли це покращить продуктивність, і щоб уникнути індексованого пошуку коли це заважатиме продуктивності. Із вищесказаного випливає, що перебудова індексів має на меті створення нових індексів або індексних груп, які точно покращать ефективність вибірки, а не навпаки. Старі індекси, що негативно впливали на продуктивність, видаляють або замінюють більш ефективними [20, 21].

Проте слід пам’ятати, що продуктивність вставки та оновлення буде погіршена, якщо використовується велика кількість індексів. Таким чином необхідно враховувати специфіку програмної системи, відносно якої проводиться редизайн сховища даних.

Індексування може виявитись недостатньо для підвищенні продуктивності. Часто існують дані, які можна не враховувати при кожному запиті, а скористатись денормалізацією, тобто замість декомпозиції даних у нові таблиці (нормалізація), виконувати зворотній процес. Часто це може допомогти уникнути додаткової

алгоритмічної складності, яка буде негативно впливати на продуктивність програмної системи.

Нормалізація була і залишається правилом, яким слід керуватися під час виконання процедур проектування бази даних. Після Б. Кодда велика кількість науковців досліджували нормальні форми та процес нормалізації. Упорядковуючи дані в стандартних форматах, нормалізація спрямована на усунення аномалій оновлення та підвищення доступності даних. Хоча вона має багато переваг і фактично вважається найкращою практикою для проектування реляційної бази даних, вона має принаймні один суттєвий недолік, а саме низьку продуктивність системи. Така низька продуктивність може серйозно вплинути на продуктивність, а тому денормалізація стала популярним рішенням у рамках проведення редизайну сховища даних [22].

Однією із технік денормалізації є метод руйнування зв'язків, що полягає у відмови від подальшої нормалізації, а отже і створенні проміжних таблиць при побудові відношень «Один-до-багатьох», або «Багато-до-багатьох». Для прикладу було взято таблиці user – користувач, post – публікація користувача, post_category – категорії, до яких належить конкретна публікація, category – можливі категорії публікацій. При цьому, зазвичай кількість категорій статична та не досить велика. Стандартна ER-діаграма при дотриманні правил нормалізації виглядала б, я на рисунку 2.1.

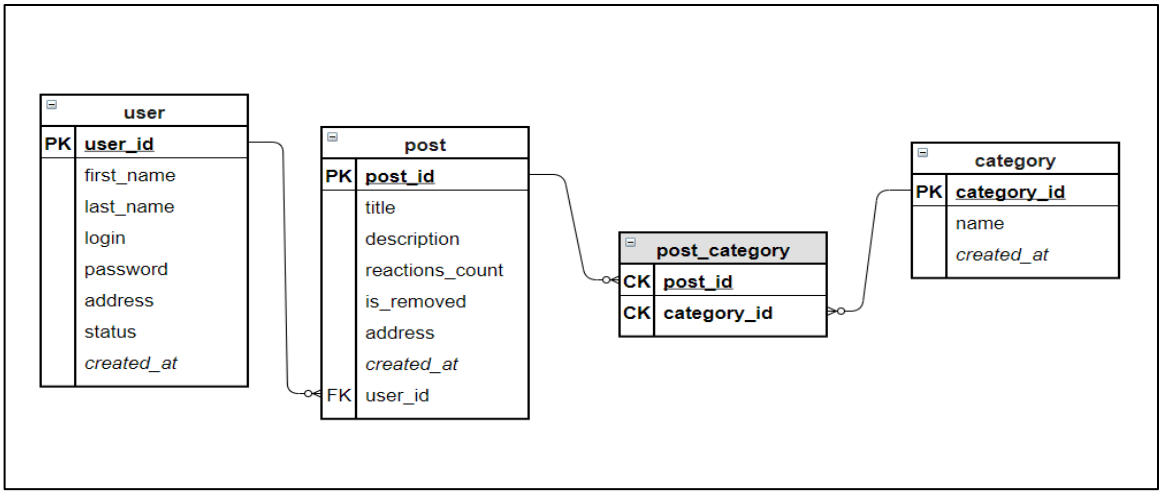


Рисунок 2.1 – Типова ER-діаграма зі зв’язком «Багато-до-багатьох»

Як було зазначено, кількість категорій та категорій, до яких може належати публікація, у типових веб-додатках зазвичай не велика, а тому при такому випадку використовувати проміжну таблицю (на рис. 2.1 – post_category) недоцільно. Оскільки кількість публікацій є необмежено великою, то і таблиця post_category буде мати необмежено велику кількість даних, що може вплинути на продуктивність вибірки при об'єднанні таких таблиць. Окрім того, при наявності таких таблиць база даних буде важити більше, аніж без них.

Застосувавши метод руйнування зв'язків, можна вирішити дану проблему. Для цього необхідно видалити проміжну таблицю, а дані, що знаходяться в ній, денормалізувати у форму строкового типу та вставити у таблицю post у якості додаткового стовпця categories. У деяких СКБД (наприклад, PostgreSQL) є підтримка масивів, а також більш складних об'єктів, таких як JSON. На рисунку 2.2 показано використання методу руйнування зв'язків.

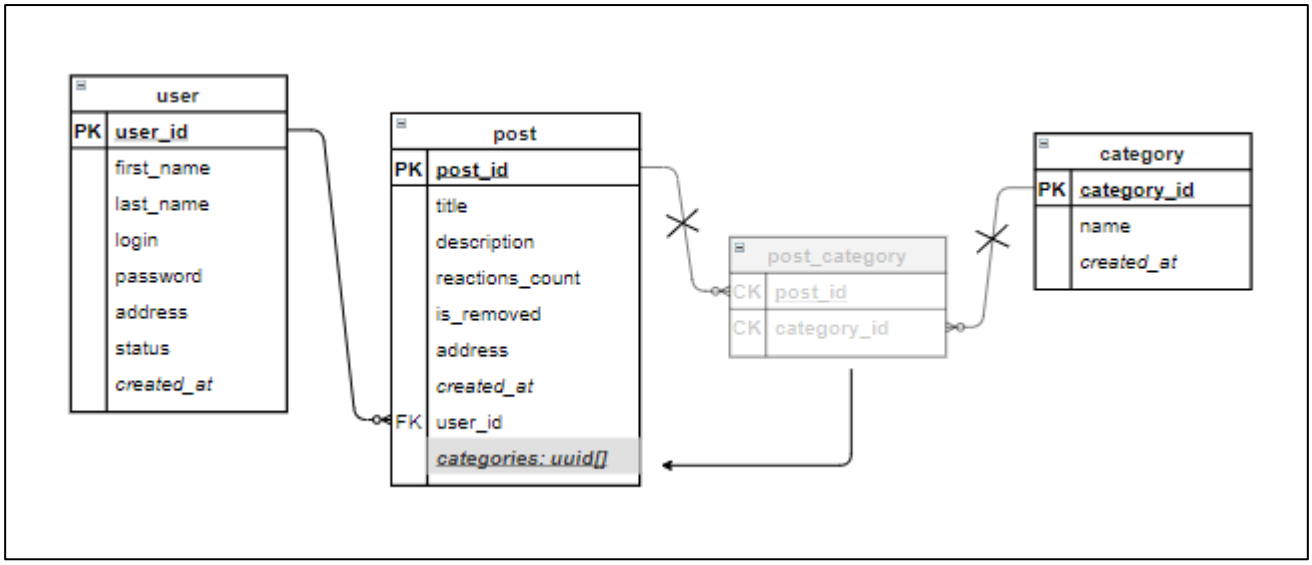


Рисунок 2.2 – Реалізація методу руйнування зв'язків

При цьому у таблиці post у новоствореному стовпці містяться унікальні ідентифікатори категорій, що відповідають записам з таблиці category. У результаті проведеної оптимізації відбулось уникнення об'єднання із проміжною таблицею із великою кількістю даних, а отже відбувся приріст продуктивності.

Проте такий підхід не буде дієвим, якщо відомо, що у проміжних таблицях може бути нескінченно велика кількість даних, що постійно змінюються. Оскільки забагато вмісту в одному стовпці у базі даних може викликати проблеми з продуктивністю програми.

Продуктивність бази даних також можна підвищити шляхом мінімізації кількості звернень до даних у вторинному сховищі після обробки транзакцій. Розбиття відношення у сховищах даних і вітринах даних є важливим кроком у проектуванні бази для меншої кількості звернень до диска. Коли окремі частини відношення використовуються різними програмами, воно може бути розділене на кілька для кожної групи обробки. Відношення можна розділити вертикально або горизонтально. Це процес поділу логічного відношення на кілька фізичних об’єктів, при якому отримані фрагменти файлу складаються з меншої кількості атрибутів а, отже, виконується менше доступів до вторинного сховища для обробки транзакції. Цей метод денормалізації називається Partitioning relation та дозволяє підвищити продуктивність [22]. Єдиним недоліком є впровадження додаткової підтримки апаратних засобів для реалізації такого підходу.

Ще один підхід, який може застосовуватись у редизайні сховища даних – це матеріалізація даних, що означає кешування результатів запиту до бази даних. Реалізація відбувається за рахунок використання спеціального об’єкту бази даних – матеріалізованих представлень.

У будь-якій системі керування базою даних, що дотримується реляційної моделі, представлення — це віртуальна таблиця, що представляє результат запиту до бази даних. Щоразу, коли запит або оновлення звертаються до віртуальної таблиці звичайного представлення, СУБД перетворює їх у запити або оновлення базових таблиць, що лежать в основі. Матеріалізоване представлення використовує інший підхід: результат запиту кешується як конкретна («матеріалізована») таблиця (а не подання як таке), яке час від часу може оновлюватися з вихідних базових таблиць. Це забезпечує набагато ефективніший доступ за рахунок додаткової пам’яті та можливої застарілості деяких даних. Матеріалізовані

представлення знаходять застосування особливо в сценаріях сховищ даних, де часті запити фактичних базових таблиць можуть бути дорогими.

У матеріалізованому представленні індекси можна будувати на будь-якому стовпці. У звичайному, навпаки, можна використовувати лише індекси для стовпців, які походять безпосередньо з (або мають відображення) індексованих стовпців у базових таблицях.

При цьому основною перевагою матеріалізованих представлень є швидкий доступ до даних за рахунок відсутності складних операцій, таких як об’єднання таблиць тощо. Але недоліком є те, що оновлення зазвичай відбувається шляхом повторного виконання вихідного запиту, результат якого був закешований у вигляді представлення. Тому метод матеріалізації зазвичай використовують при доступі до архівних даних, а застосування матеріалізованих представлень на постійній основі може призвести до проблем з продуктивністю, оскільки їх оновлення займає багато часу, а також у деяких системах керування базами даних відбувається блокування таблиць.

Одним з методів підвищення швидкості оновлення матеріалізованих представлень є метод інкрементного оновлення. При перезаписі враховуються лише зміни у вихідних таблицях. Часто для цього використовуються різні структури даних, як, наприклад, LSM-дерево, яке є добре відомою структурою, прийнятою для покращення продуктивності запису [23]. Перевагами інкрементного оновлення є краща продуктивність перезапису матеріалізованих представлень за рахунок вибіркового застосування змін. Проте, не усі реляційні системи підтримують такий підхід. Крім того, процес оновлення матеріалізованих представлень блокує таблиці для запису, що може негативно вплинути на швидкість вставки та оновлення.

Матеріалізовані представлення використовуються для обчислення об’єднання, агрегації та, якщо це можливо, можуть значно скоротити час виконання запиту. Щоб забезпечити правильний результат, матеріалізоване представлення має бути актуальним щоразу, коли до нього звертаються за допомогою запиту. Більшість систем баз даних досягають цього завдяки

ретельному технічному обслуговуванню, де всі задіяні представлення зберігаються як частина оператора оновлення або транзакції оновлення.

За умови методу Eager Maintenance витрати на підтримку представлення повністю покриваються оновленнями. Накладні витрати на технічне обслуговування представлення можуть бути досить високими, коли воно здійснюється до багатьох представлень, що призводить до падіння продуктивності оновлення. Це також може бути неефективним, якщо є багато невеликих оновлень у системі. Щоб вирішити цю ситуацію, деякі системи баз даних також підтримують відкладену технічну підтримку оновлення представлень, коли технічне воно відбувається лише тоді, коли користувач явно запускає його. Цей підхід має серйозний недолік: запит може мати застаріле подання та давати неправильний результат. Якщо дозволити оптимізатору запитів автоматично використовувати такі подання, це негативно відобразиться на коректності результатів вибірки. Використання матеріалізованих представлень більше не є автоматичним і прозорим для користувачів. Автори запитів повинні знати, які представлення використовуються в запиті, як вони обслуговуються та чи є вони актуальними на момент виконання.

Рішення, яке одночасно не потребує обслуговування перегляду через оновлення та зберігає актуальність представлення, називається відкладеним матеріалізованим представленням. Обслуговування такого подання представлене у статті [24], досягає цих двох суперечливих цілей. Під час відкладеного оновлення не зберігаються представлення, а лише – достатньо інформації, щоб пізніше можна було підтримувати задіяні подання. Фактичне технічне обслуговування виконується за допомогою завдань із низьким пріоритетом, коли система має доступні вільні цикли. Якщо в системі достатньо безкоштовних циклів і перегляд підтримується до того, як він стане потрібним для запитів, ні оновлення, ні запити не впливають негативно на обслуговування представлення. Якщо подання не оновлюється, коли це необхідно для запиту, воно прозоро оновлюється перед тим, як запиту буде дозволено доступ до нього. Lazy Maintenance дозволяє швидше завершувати оновлення, тому блокування знімаються швидше, що зменшує

частоту конфліктів блокувань і переривання транзакцій. Це особливо важливо для оновлень, які впливають на високоагреговані представлення, оскільки вони, як правило, мають вищий рівень конфліктів блокувань.

У публікації [24] було експериментально продемонстровано переваги відкладеного обслуговування в останньому розділі. Однак не рекомендується замінювати Eager Maintenance на Lazy Maintenance для всіх матеріалізованих представлень. Кожен підхід має свої переваги та недоліки, і який підхід є кращим для певного перегляду, залежить від програми. Загалом, вибір стратегії обслуговування для матеріалізованого перегляду залежить від таких факторів:

– співвідношення оновлень до запитів і як швидко запити посилаються після оновлень представлень;

– розмір оновлень (кількість рядків, на які впливає кожне оновлення) відносно вартості обслуговування перегляду.

Eager Maintenance підходить для матеріалізованих представлень, базові таблиці яких рідко оновлюються, і за оновленнями, ймовірно, швидко надсилатимуться запити. З іншого боку, Lazy Maintenance краще використовувати для переглядів із більш частими невеликими оновленнями, вартість обслуговування яких є відносно високою.

Ще один популярний метод редизайну бази даних – зберігання значення лічильників у вигляді окремого стовпця замість виконання операцій підрахунку кількості записів, що пропонуються реляційною базою даних. Проте основний недолік – неможливість використання даного методу для динамічного підрахунку відфільтрованих даних. Для цього іноді можна використовувати вбудовані функції, які напряму чи опосередковано надають результат, що містить кількість задіяних у запиті даних. Наприклад, якщо точна кількість не важлива, то можна використати функції EXPLAIN, як вказано у [25]. Для підрахунку даних використовується спеціальна функція, яка включає результат виконання запиту EXPLAIN. Недоліком є можливість застосування SQL-ін’єкцій та невисока точність результатів, тому такий метод використовується лише у випадках, де не потрібно знати точну кількість відфільтрованих даних.

Вищенаведені методи редизайну сховища даних необхідно проводити лише після детального аналізу та виявлення ділянок, де відбувається деградація продуктивності бази даних. Крім того, усі ці методи можуть бути складовою комплексного підходу та використовуватись разом, що може завдати значний вплив на продуктивність програмної системи в цілому.

Наступний етап – декомпозиція сховища, або міграція проводиться, якщо усі вище наведені методи редизайну не дали відчутного результату, або не задовільнили очікувані показники продуктивності бази даних.

Однією з причин може бути зберігання даних у невластивій поточній базі даних структурі. Прикладом може виступати графова структура даних та запити на знаходження глибинних зв'язків між вершинами. До прикладу можна взяти побудову відношень між користувачами, наприклад, відношення типу «підписник-спостерігач». Як правило, у реляційній базі даних для цього формують додаткову проміжну таблицю, наприклад, user_relation (рис. 2.3).

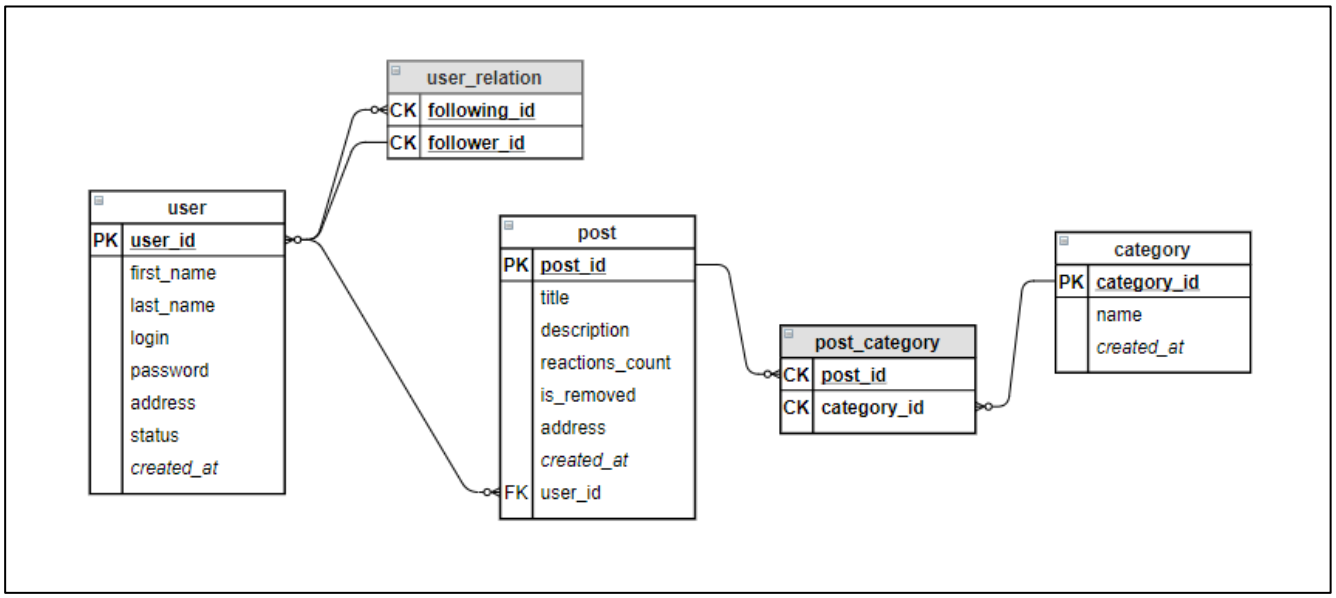


Рисунок 2.3 – ER-діаграма із проміжною таблицею user_relation

У таблиці user_relation є два зв'язки, які пов'язують її тільки із таблицею user, оскільки і підписник, і спостерігач – це користувач, а отже, відноситься до таблиці user. Це означає, що при вибірці із таблиць, буде відбуватись подвійне об'єднання

із таблицею user. Така поведінка об’єднання має назву self-join, що означає об’єднання сама з собою, тобто два набори даних в одній таблиці об’єднуються між собою. Проте у вищенаведеному випадку об’єднання відбувається сам з собою проте через проміжну таблицю, що є виродженим випадком self-join-у. Під час того, як користувач підписується на іншого, дані заносяться у проміжну таблицю. Користувач, хто підписався, записується у стовпець follower_id, а той, на кого підписаний – following_id.

Якщо застосовувати усі вище наведені техніки редизайну до такої схеми бази даних, то можна покращити продуктивність. Наприклад, застосувавши реіндексацію, вибірка даних з об’єднанням буде відбуватись швидше.

Проте, наприклад, застосувати денормалізацію, використовуючи метод руйнування зв’язків, як у табличці post_category, оскільки одразу невідома кількість підписників у користувача, тобто їх кількість є динамічною і може бути як завгодно великою. Крім того, один користувач може мати як підписників, так і сам бути підписником, що ускладнює ситуацію теоретичного використання масивів, як типу стовпця.

Використання матеріалізації теоретично могло б вирішити проблему виконання складних запитів, проте матеріалізовані представлення необхідно часто оновлювати, а інкрементні чи відкладені оновлення підтримуються не у всіх сховищах даних. А часте виконання таких складних запитів, як вибірка публікацій підписників займе багато часу, що негативно вплине на продуктивність при великій кількості даних.

Для тестування було створено тестову схему даних та згенеровано близько 100 тис. записів у таблиці user, та близько 10 м. – у таблиці post. Виконано запит вибірки публікацій усіх користувачів, на які підписаний даний користувач, відсортованих по даті створення публікації, та встановлено ліміт на вибірку у 1 тис. записів. Сам запит виглядає таким чином:

```
SELECT
    DISTINCT
      "user"."user_id",
      "user"."first_name",
      "user"."last_name",
      "post"."post_id",
      "post"."title",
      "post"."description",
      "post"."reactions_count",
      "post"."created_at"

      FROM "user"
      JOIN "user_relation" on "user"."user_id" =
           "user_relation"."following_id"
      JOIN "post" on "user_relation"."following_id" = "post"."user_id"
      WHERE
            "user_relation"."follower_id" =
               '419c2a96-930c-456f-b624-97c27971377e'
       AND
           "post"."created_at" <= timezone('utc'::text, now())
      GROUP BY "user"."user_id", "post"."post_id"
      ORDER BY "post"."created_at" DESC
      OFFSET 0 LIMIT 1000;
```

Результат вибірки представлено на рисунку 2.4.

	user_id uuid	first_name character varying (100)	last_name character varying (100)	post_id uuid	title character varying (500)	description character varying (2000)	reactions_count numeric	created_at timestamp without time zone
1	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	19e15227-a96d-41b9-b19a-57f74185ef7f	Title 330b4aed346399...	Description e1972b7ed6...	0	2022-11-13 13:42:55.332885
2	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	1f5155c2-a773-46dc-aa45-2a198815ae...	Title 4e1216a7b62d5ef...	Description ab8a4aeeb4...	0	2022-11-13 13:42:55.332885
3	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	3647109b-b2cd-4c37-9d66-3124f5c986...	Title 3aad5f2eda8712e...	Description 8ccf2b7ea2...	0	2022-11-13 13:42:55.332885
4	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	5a1cce9c-32fc-4b3f-8e02-378fc66a11d6	Title ef51fac8c77f02d...	Description 37e0b4043...	0	2022-11-13 13:42:55.332885
5	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	9e2723e3-887c-4adc-a4f8-e9a001e629...	Title f1057cfda7563f3...	Description 0c9fcb4917...	0	2022-11-13 13:42:55.332885
6	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	cd105c1e-bda6-4cb9-abf0-2d90214842...	Title 244034ef893cbe0...	Description ce659bd646...	0	2022-11-13 13:42:55.332885
7	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	cecf7eb8-ace7-4edd-a658-62bc5f9fb974	Title e43a428c7b3395...	Description 6fa30aef66...	0	2022-11-13 13:42:55.332885
8	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	e446df8c-74b1-45cd-a357-a8f2225e40...	Title 1b1ee3d4f96e832...	Description 3c6159f326...	0	2022-11-13 13:42:55.332885
9	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	e7b970be-0159-4516-bd18-9c7ab6922...	Title 5c5a629451e353f...	Description 035242162...	0	2022-11-13 13:42:55.332885
10	000122ce-38ae-42eb-a47b-650416102060	First Name 5334	Last Name 5334	f613add7-012f-4201-ad75-b3f1624f292c	Title d1996ec2b62927...	Description 55f1f5dc0c...	0	2022-11-13 13:42:55.332885
11	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	26da9b13-b89f-46fe-82b9-54a3d5c3e9e8	Title 041b84fd4b7e4ec...	Description 0f030c710d...	0	2022-11-13 13:42:55.332885
12	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	2c8d87ab-77cb-4d54-811d-06e886a532...	Title 5fd1a4fa47acfeb...	Description c593aa24d2...	0	2022-11-13 13:42:55.332885
13	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	2e167eb4-54c4-45b6-994b-1adc683f6f...	Title 09ec2e1b4e1771...	Description 41c65eee9f...	0	2022-11-13 13:42:55.332885
14	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	3bf05c6b-4387-41d9-becf-a0ab16cc4fb6	Title 7209a8655abf40f...	Description 39a11e3906...	0	2022-11-13 13:42:55.332885
15	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	4912d2ea-05fa-4e89-9372-f9471be68e...	Title f2418d8048318bf...	Description c5a75a752b...	0	2022-11-13 13:42:55.332885
16	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	5468b291-b3c9-43c8-b6da-d8602ef354...	Title 023b99618d4368...	Description c3ecfad014...	0	2022-11-13 13:42:55.332885
17	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	63dc1432-8ea1-48f5-892b-85fac80874...	Title f4fe96651e7ef07...	Description 68288b791...	0	2022-11-13 13:42:55.332885
18	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	9e806149-566f-4b9a-af0f-f4272092ecb1	Title bc75836a7e0feb0...	Description e51a66f4b2...	0	2022-11-13 13:42:55.332885
19	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	a5c49d7e-ad64-41e9-bf62-9689e0b774...	Title 747183846afd60...	Description 5edc1f4015...	0	2022-11-13 13:42:55.332885
20	0001d70d-bb77-4f85-a0a7-a6289ba00cb5	First Name 20240	Last Name 20240	cdf3f40a-cacd-4175-90fa-1588234c3156	Title 2a7563871bc367...	Description a01c806dc7...	0	2022-11-13 13:42:55.332885
21	0002e01e-fd9c-4ea3-a3bd-ca1da3af5582	First Name 45257	Last Name 45257	0ed11221-f055-4b92-8b24-9b60e8a863...	Title f084e7d634bcd41...	Description 0cdc360dc9...	0	2022-11-13 13:42:55.332885
22	0002e01e-fd9c-4ea3-a3bd-ca1da3af5582	First Name 45257	Last Name 45257	28336f40-3b22-42ab-a4ef-dd75b54b17ff	Title bcdba3b4f9f8c97...	Description 3faaf72780...	0	2022-11-13 13:42:55.332885
23	0002e01e-fd9c-4ea3-a3bd-ca1da3af5582	First Name 45257	Last Name 45257	3aec520a-4d82-4c93-b945-b1d43622a8...	Title cc7862ea61cb34...	Description a2f2403fbd...	0	2022-11-13 13:42:55.332885
24	0002e01e-fd9c-4ea3-a3bd-ca1da3af5582	First Name 45257	Last Name 45257	4ab1b92e-19bd-4f29-b510-6f9c996495...	Title 5167daae765eb4...	Description 42a918d82c...	0	2022-11-13 13:42:55.332885
25	0002e01e-fd9c-4ea3-a3bd-ca1da3af5582	First Name 45257	Last Name 45257	4b77ae7c-0431-47f7-a107-43594c1a72...	Title 7b0b71bdd845a6...	Description 5962021ea1...	0	2022-11-13 13:42:55.332885

Рисунок 2.4 – Результат виконання запиту вибірки публікацій підписників користувача

Як видно, запит виконався коректно, дата створення публікацій відсортована у порядку спадання, а також згруповано публікації відносно їх авторів. Проте є недолік, який полягає у дуже довготривалій роботі подібних запитів. План

виконання, що наведено на рисунку 2.5, показує велику кількість послідовних сканувань та утворення вкладених об’єднань, що говорить про нездатність оптимізатора виконати покращення запиту, навіть враховуючи наявність індексів на ключових та інших необхідних стовпцях.

	QUERY PLAN text
2	-> Unique (cost=744490618.71..744491518.71 rows=40000 width=148) (actual time=9437.383..9439.187 rows=1000 loops=1)
3	-> Sort (cost=744490618.71..744490718.71 rows=40000 width=148) (actual time=9437.382..9438.965 rows=1000 loops=1)
4	Sort Key: post.created_at DESC, "user".user_id, "user".first_name, "user".last_name, post.post_id, post.title, post.description, post.reactions_count
5	Sort Method: external merge Disk: 306640kB
6	-> Group (cost=744467603.62..744487561.16 rows=40000 width=148) (actual time=3838.551..4889.472 rows=2000020 loops=1)
7	Group Key: "user".user_id, post.post_id
8	-> Gather Merge (cost=744467603.62..744486761.16 rows=160000 width=148) (actual time=3838.549..4352.046 rows=2000020 loops=1)
9	Workers Planned: 4
10	Workers Launched: 4
11	-> Sort (cost=744466603.56..744466703.56 rows=40000 width=148) (actual time=3767.917..3848.325 rows=400004 loops=5)
12	Sort Key: "user".user_id, post.post_id
13	Sort Method: external merge Disk: 63816kB
14	Worker 0: Sort Method: external merge Disk: 61608kB
15	Worker 1: Sort Method: external merge Disk: 60840kB
16	Worker 2: Sort Method: external merge Disk: 61016kB
17	Worker 3: Sort Method: external merge Disk: 59352kB
18	-> Partial HashAggregate (cost=744463146.02..744463546.02 rows=40000 width=148) (actual time=3022.248..3178.597 rows=400004 loops=5)
19	Group Key: "user".user_id, post.post_id
20	-> Nested Loop (cost=0.84..114306744.52 rows=126031280300 width=148) (actual time=0.246..2709.774 rows=400004 loops=5)
21	Join Filter: (user_relation.following_id = "user".user_id)
22	-> Nested Loop (cost=0.42..333403.36 rows=252057520 width=133) (actual time=0.181..1651.821 rows=400004 loops=5)
23	-> Parallel Seq Scan on post (cost=0.00..96682.09 rows=500005 width=117) (actual time=0.024..314.664 rows=400004 loops=5)
24	Filter: (created_at <= timezone('utc'::text, now()))
25	-> Index Only Scan using user_relation_pkey on user_relation (cost=0.42..0.46 rows=1 width=16) (actual time=0.003..0.003 rows=1 loops=2000020)
26	Index Cond: ((follower_id = '419c2a96-930c-456f-b624-97c27971377e'::uuid) AND (following_id = post.user_id))
27	Heap Fetches: 2000020
28	-> Index Scan using user_pkey on "user" (cost=0.42..0.44 rows=1 width=47) (actual time=0.002..0.002 rows=1 loops=2000020)
29	Index Cond: (user_id = post.user_id)
30	Planning Time: 0.515 ms
31	Execution Time: 9482.649 ms

Рисунок 2.5 – План запиту

Також видно, що тривалість виконання запиту сягає близько 9 секунд.

Отже, якщо редизайн сховища даних не вирішив проблему із продуктивністю, що означає недосягнення очікуваних показників (зазвичай,

очікуваними та високими показниками продуктивності насьогодні вважається тривалість відповіді на запит користувача < 1 секунди).

Як видно, структура подання даних є невластивою для реляційної моделі. Тому необхідно перейти до наступного етапу реінжинірингу сховища даних, що називається міграцією. Один з методів – це міграція усієї схеми даних на інший тип сховища. Зв’язки між користувачами дуже зручно виділити у вигляді орієнтованого графа, а тому міграція відбуватиметься у графову базу даних. Для тестових цілей було обрано СКБД Neo4j.

Існує кілька областей застосування, у яких дані мають природне представлення у вигляді графу. Це відбувається, наприклад, у соціальних мережах, геолокаційних застосунках тощо. У цьому контексті реляційні системи зазвичай непридатні для зберігання таких даних, оскільки вони навряд чи охоплять притаманну їм графову структуру. Крім того, і що важливіше, обхід графа над високо зв’язаними даними потребує складних операцій з’єднання, що може зробити типові операції з такими даними неефективними, а програми – важко масштабованими. У GDBMS дані зберігаються у вигляді графів, а запити виражаються в термінах операцій їх обходу. Це дозволяє програмам масштабуватися до дуже великих наборів даних. Крім того, оскільки GDBMS не покладаються на жорстку схему, вони забезпечують більш гнучке рішення в сценаріях, коли схема даних швидко розвивається. У цій структурі міграція постійного рівня програми з реляційної системи зберігання на основі графів може бути дуже корисною.

У статті [26] запропонована методологія перетворення реляційної бази даних у, використовуючи наявну схему, що дозволить скоротити час на міграції усієї реляційної схеми. Крім того запропоновані методи дозволяють використовувати лише частину реляційної схеми, та при певному удосконаленні – можуть забезпечити зручний інтерфейс управління міграціями з реляційного сховища до графової бази.

Отже, при міграції у графовий тип, було створено схему даних, яку подано на рисунку 2.6.

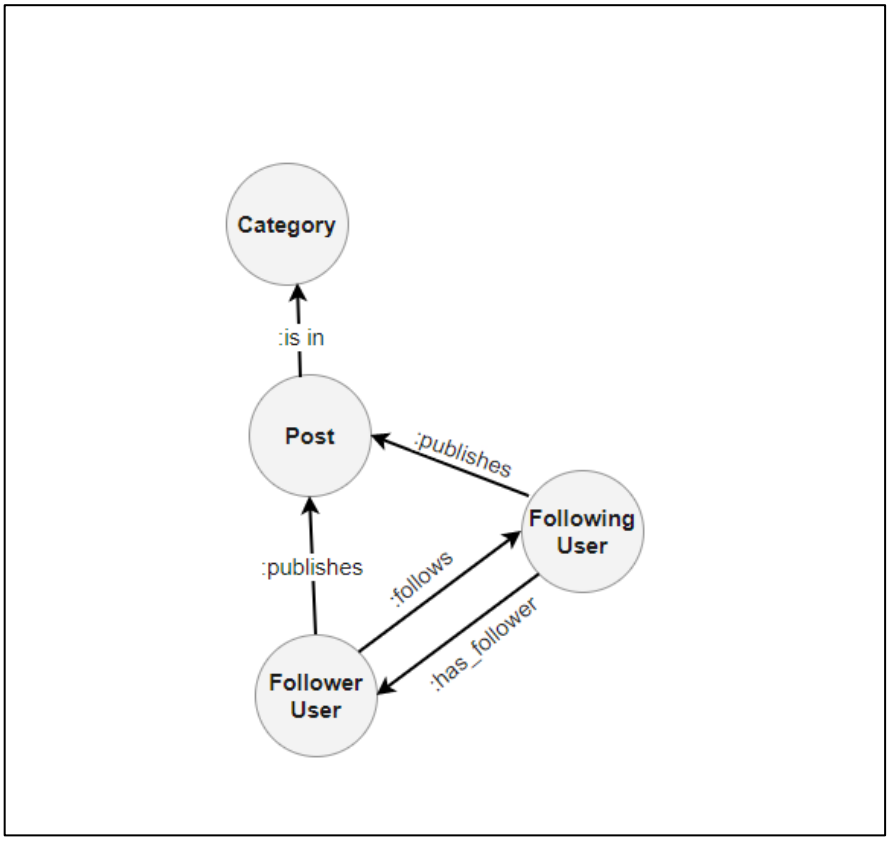


Рисунок 2.6 – Графова схема, утворена при міграції реляційної бази даних

Після міграції таблиці реконструйовано у вузли графа, а відношення між таблицями – його ребра. Кожне відношення та вузол має назву, а також поля, що відповідають властивостям об’єкта. При цьому для даної графової було також згенеровано аналогічна до реляційної бази даних кількість вузлів користувачів, публікацій, а також відношень між ними. Запит вибірки публікацій усіх користувачів, на які підписаний даний користувач, відсортованих по даті створення публікації приведено у лістингу:

```
MATCH (u1:User)-[:Follows]->(u2:User)-[:Publishes]->(p:Post) where ID(u1)= 2
return distinct u1, u2, p Limit 1000
```

При цьому результат виконання запиту буде аналогічним до того, який було отримано у реляційній БД, але у другому форматі. Він поданий на рисунку 2.7.

"p"
{"reactions_count":0,"description":"Description 0.19868936787983793","is_removed":false,"title":"Title 0.10611094136021026"}
{"reactions_count":0,"description":"Description 0.7499347763015751","is_removed":false,"title":"Title 0.16718103299612308"}
{"reactions_count":0,"description":"Description 0.981257642859991","is_removed":false,"title":"Title 0.20915295172000625"}
{"reactions_count":0,"description":"Description 0.7606704058632275","is_removed":false,"title":"Title 0.33917987502461544"}
{"reactions_count":0,"description":"Description 0.7901056938450943","is_removed":false,"title":"Title 0.3711367671995145"}
{"reactions_count":0,"description":"Description 0.8125770010573489","is_removed":false,"title":"Title 0.025456404740945016"}

Рисунок 2.7 – Результат виконання запиту вибірки публікацій у графовій базі

Також можна визначити план запиту за допомогою спеціальних команд, які представляє СКБД Neo4j. Він поданий на рисунку 2.8.

Рисунок 2.8 – Частина плану виконання запиту вибірки публікацій

Згідно рис. 2.8 – час виконання вищенаведеного запиту склав всього приблизно 29 мілісекунд, що значно швидше, ніж виконання подібних запитів у реляційній базі даних. А це означає, міграція з одного типу на інший позитивно вплинула на продуктивність програмної системи. У такому разі зазвичай проводять додаткові навантажувальні тести, а також тести сумісності. Після успішного виконання процес редизайну вважається завершеним, оскільки продуктивність досягла рівня очікуваної.

Переваги очевидні, проте є і недоліки. В більшості випадках міграція на новий тип сховища даних може бути більш затратним процесом, оскільки, по-перше, вартість хостингу серверів баз даних іншого типу може бути вищою в силу різних факторів, а по-друге, команді розробників та dba-інженерів необхідний час для навчання на нових технологіях. Крім того, перенесення завжди вимагає повного тестування усієї програмної системи після завершення міграції.

Якщо схема даних великих розмірів, то можна використати технологію мультимодельних баз даних. Мультимодельним сховищем даним виступає платформа обробки даних, яка підтримує кілька моделей даних, що визначають параметри організації та впорядкування інформації в базі даних. Можливість об’єднання кількох моделей в одну базу даних дозволяє задовольняти різноманітні вимоги додатків без необхідності розгортати різні системи баз даних.

Моделі даних, які можуть використовувати такі бази даних, включають реляційні, ієрархічні та об’єктні, а також графові структури, wide-column-тип та ключ-значення, які об’єднані разом та взаємодіють між собою як одне ціле. Окремі постачальники баз даних пропонують різні комбінації моделей даних.

На відміну від систем, які обмежуються виключно провідною на ринку реляційною моделлю даних, мультимодельні бази не однаково зберігають дані в структурі таблиці на основі рядків. У результаті вони можуть обробляти різні форми даних, які не відповідають фіксованій попередньо спроектованій схемі реляційної моделі, включаючи неструктуровані та напівструктуровані типи даних. Проте в деяких випадках мультимодельний підхід може обмежити транзакційну

цілісність, яку використовують системи управління реляційними базами даних для підтримки точності та узгодженості даних [27].

Згідно з архітектурним рішенням популярної соціальної мережі Instagram, відношення user_relation могло б виступати у якості wide-column-бази даних (наприклад, Cassandra), повнотекстовий пошук публікацій – документного сховища (наприклад, Elastic Search), а решта – реляційної структури (у PostgreSQL) [6]. Крім того, мультимодельні бази даних надають уніфікований інтерфейс для взаємодії із гетерогенними даними, що дозволяє зекономити час написання запитів та фінансові витрати на фахівців, які мають досвід у різних технологіях. На рисунку 2.9 зображена узагальнена мультимодельна схема, що є результатом другого етапу реінжинірингу – декомпозиції та міграції.

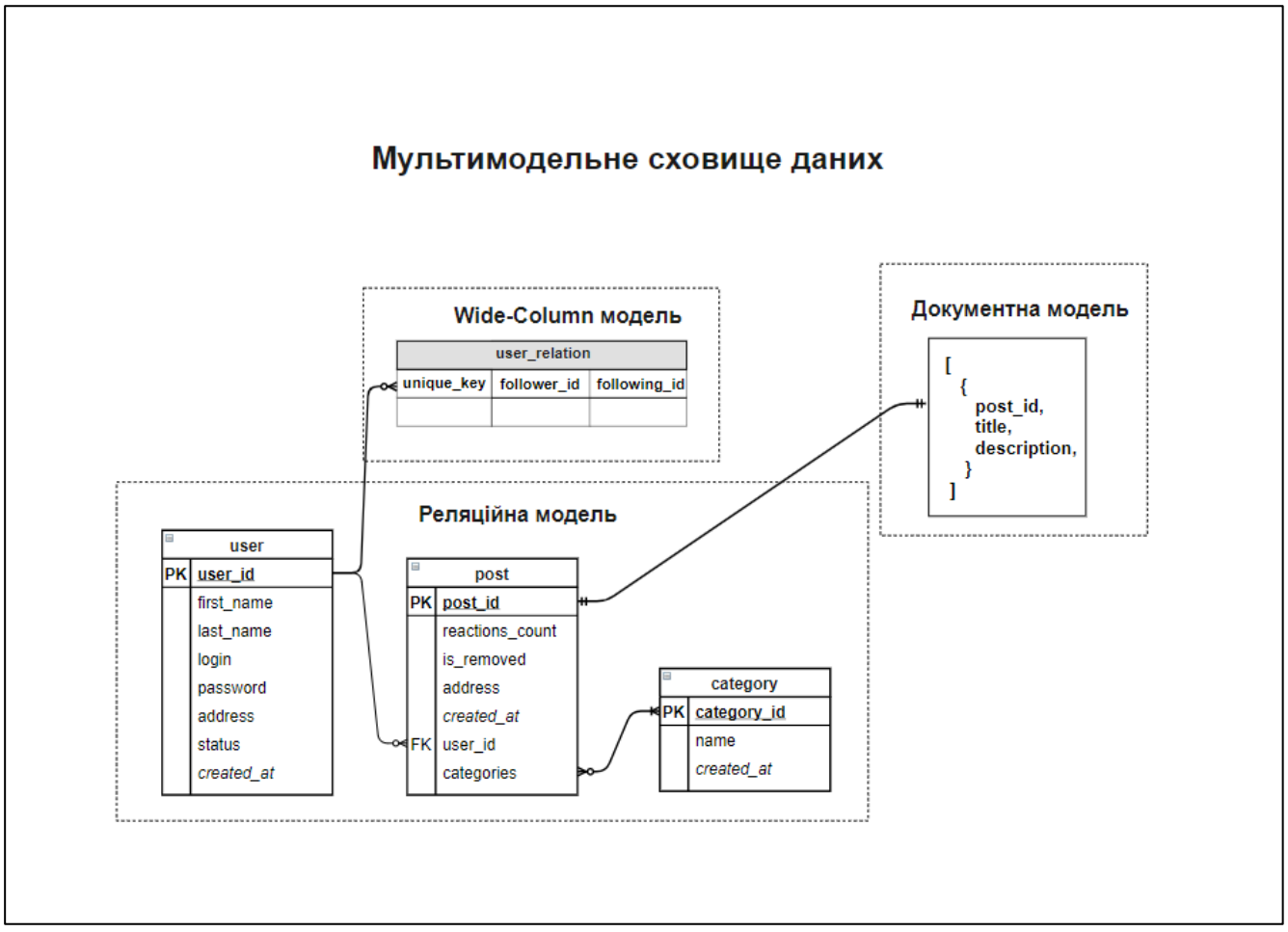


Рисунок 2.9 – Мультимодельне сховище даних

Кожна мультимодельна база даних має технологію взаємодії зі різними моделями даних у вигляді проміжного зв’язного програмного забезпечення. Так, наприклад, у схемі, що зображена на рисунку 2.9, можливе об’єднання даних між wide-column, реляційною та документною моделями даних. При цьому це майже не буде відрізнятись від класичного реляційного об’єднання декількох таблиць. Але перевага у тому, що усі таблиці належать до властивим їм структурами. Це підвищує продуктивність програмної системи.

Проте, через незрілість таких технологій виникають певні ризики у застосуванні їх на ринку, а підтримка таких рішень вимагає багато фінансових ресурсів та апаратних засобів, щоб показники продуктивності були допустимими. Такий основний недолік пояснює непопулярність використання мультимодельних програмних систем у корпоративних програмних системах.

Таким чином, бачимо, що наявні методи вирішення проблем реінжинірингу бази даних є достатньо ефективними, проте мають вагомі недоліки, які часто вирішуються додаванням у систему нових апаратно-програмних засобів, таким чином, горизонтально масштабуючись задля рівномірного розподілення навантаження при виконанні великої кількості запитів. Етап редизайн, що передбачає зміни лише у рамках одного сховища даних не завжди може виявитися дієвим та досягти очікуваних результатів. Міграція та декомпозиція сховища може зіграти важливу роль у вирішенні проблеми падіння продуктивності, а перенесення схеми на мультимодельний тип сховища – взагалі у довгостроковій перспективі вирішує проблеми зберігання гетерогенних даних.

Проте, існують випадки, коли лише декілька запитів або таблиць негативно впливають на продуктивність і виконувати міграцію усієї структури недоцільно та неекономно, а мігрувати окремі логічні частини у інші сховища, або інші типи сховищ вимагає збільшення фінансування проекту в першу чергу через наявність нових серверних компонентів, потім – через необхідність у кваліфікованих працівниках конкретної спеціалізації що також є невигідним у корпоративних програмних системах.

Тому проблему, яка полягає у збільшенні швидкості запитів до гетерогенних структур, можна вирішувати наявними способами. Стратегія матеріалізації відкриває багато можливостей у швидкому доступі до даних, крім того, вона є частиною етапу редизайну, що означає відсутність додаткових апаратно-програмних засобів для підтримки різнорідних сховищ даних, а отже, і меншу витрату фінансових ресурсів на їх супровід.

Отже, проблему реінжинірингу бази даних можна вирішити, удосконаливши метод матеріалізованих представлень та доопрацювавши його використання таким чином, щоб була можливість використовувати матеріалізацію даних на постійній основі, що, в свою чергу, забезпечить підвищення продуктивності запитів вибірки.

2.2 Удосконалення методу матеріалізованих представлень

Метод матеріалізованих представлень полягає у тому, що при виконанні запитів до бази даних, вибірка відбувається не з таблиць схеми БД, а з спеціальних об’єктів бази даних – матеріалізованих представлень, що зберігають результат виконання запиту. Концепція матеріалізації представляє собою побудову представлень відповідно до запитів користувача. На відміну від фіксованої архітектури схеми БД, матеріалізовані представлення проектуються динамічно. Їх особливість полягає у тому, що представлення містять закешовані дані, а отже, при кожному оновленні вихідних таблиць, які пов’язані з представленнями, дані необхідно оновлювати. Звісно, це не обов’язкова дія, якщо необхідно отримати дані із, наприклад, архівних таблиць, де запити надзвичайно рідко надсилаються для отримання історичних даних.

У розділі 2.1 було наведено методи матеріалізації у рамках редизайну сховища даних, звідки можна підсумувати, що упродовж багатьох років методи підвищення швидкості оновлення представлення розроблялись та досліджувались нові шляхи забезпечення швидкого оновлення. Наприклад, інкрементне оновлення полягає у застосуванні змін до представлення, не виконуючи вихідний запит. Це

значно прискорює швидкість виконання та певною мірою дозволяє використовувати матеріалізовані подання на постійній основі. Проте, досі не усі СКБД підтримують такий підхід до оновлення представлень, а ті, що імплементують даний метод, мають не досить досконалу структуру та механізми.

Тому у дипломній роботі запропоновано розглянути матеріалізовані представлення під іншим кутом, а саме, замість удосконалення оновлення представлення використовуватимуться допоміжні таблиці за рахунок застосування принципів стратегій кешування Write Back та Read Though. При цьому, таблиці виступатимуть додатковим кешуючим шаром для забезпечення стабільної вибірки постійно оновлених даних.

Якщо порівнювати операції вставки, оновлення та видалення, то відносно часта буде перша. У більшості систем вставка є найчастішою операцією після вибірки даних. Тому, нехай, у системі є дві таблиці T1 та T2. Вважатимемо їх вихідними, оскільки по ним будується запит для вибірки даних. Нехай ці таблиці будуть зв’язані відношенням «один-до-багатьох» (рис. 2.10).

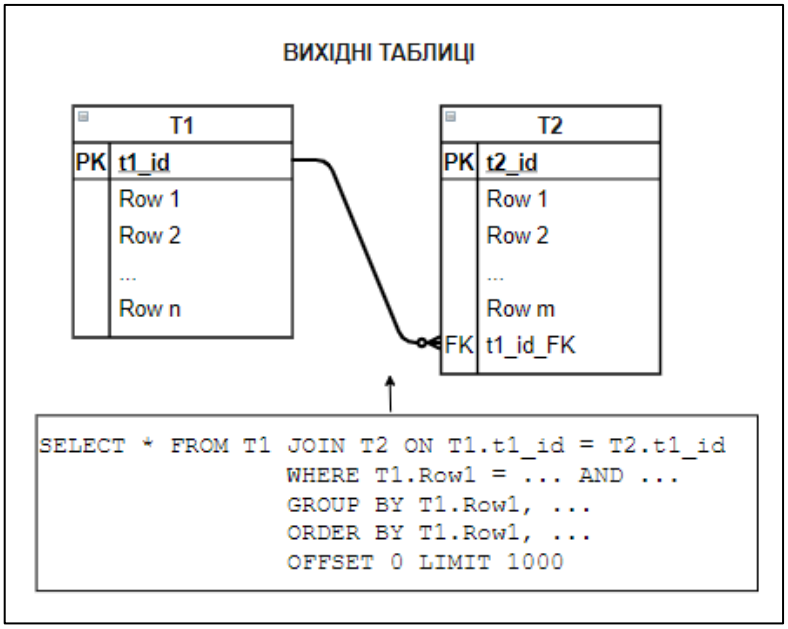


Рисунок 2.10 – Вихідні таблиці T1 та T2

При цьому, нехай, кількість даних таблиці T1 становитиме N_1, а T2 – N_2 відповідно. Припустимо, що $N_1 \gg 100$, $N_2 \gg 100$, тобто кількість у вихідних

таблицях даних досить велика. Зазвичай для прискорення вибірки із таблиць з великою кількістю даних встановлюють індекси на зовнішні ключі, що дозволяє виконувати вибірку за лінійний час, уникаючи вкладених запитів.

Асимптотична складність алгоритму вибірки у кращому випадку при застосуванні техніки Sort-Merge за умови унікальності даних у двох таблицях становитиме:

$$O(N_1 + N_2), \tag{1}$$

де O – нотація Ландау;

N_1, N_2 – обсяг першої та другої вихідної таблиці відповідно [28].

Проте, не завжди слід аналізувати лише кращі випадки, оскільки можливі запити, де не можна усі дії виконати за кращим асимптотичним сценарієм. Тому, слід звернути увагу і на гірший сценарій – при відсутності індексів механізм Sort-Merge не виконуватиметься, а матимуть місце звичайні об’єднання, що відповідно до термінології реляційних СКБД називаються Nested Loop, а вибірка з однієї таблиці при відсутності індексів – Sequence Scan. Тому, асимптотична складність у гіршому випадку становитиме:

$$O(N_1 \cdot N_2) \tag{2}$$

Тому, у випадку (2) при такій великій кількості даних швидкість виконання запиту суттєво знизиться, якщо враховувати, що запит може складатись із багатьох підзапитів, об’єднань та групувань. У випадку (1), ситуація буде краща, проте лише там, де правильно розставлено індекси.

Виходячи із вищесказаного, було прийнято рішення ввести у систему матеріалізоване представлення MV, що зберігатиме результат виконання складних запитів задля підвищення продуктивності запитів вибірки. Тепер система виглядатиме так, я подано на рисунку 2.11.

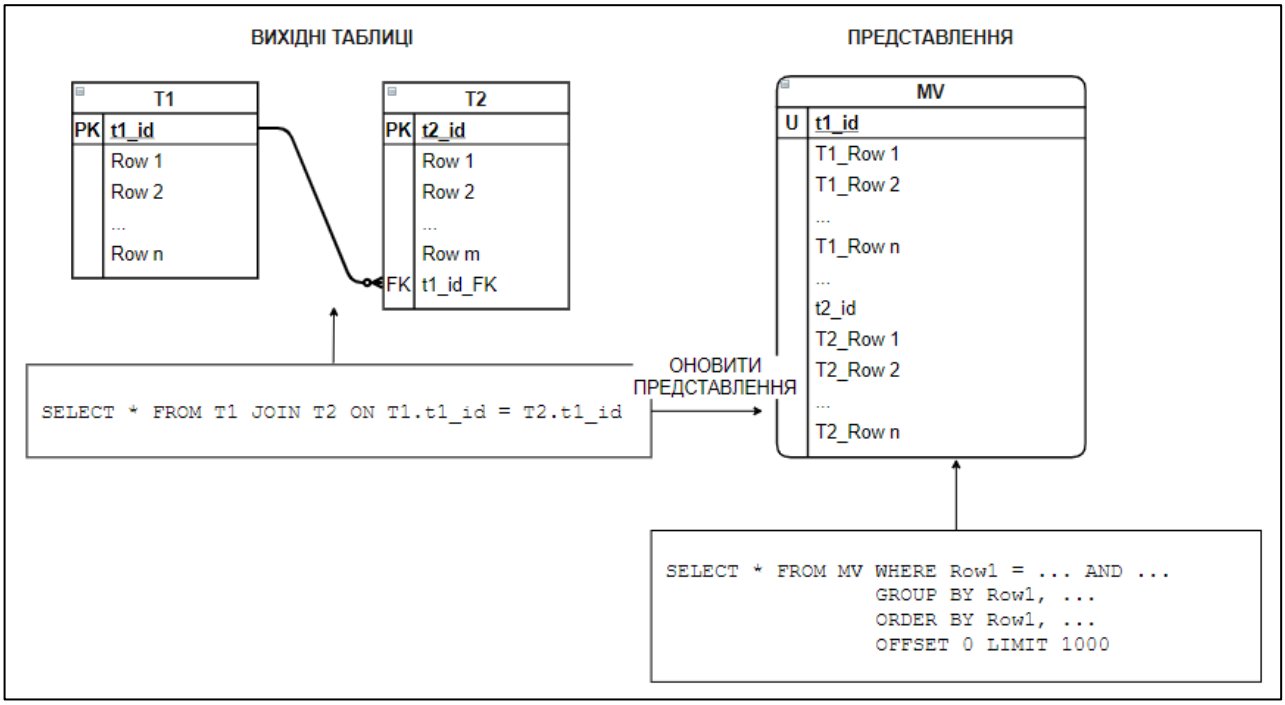


Рисунок 2.11 – Застосування матеріалізованого представлення

Кількість даних у матеріалізованому представленні буде рівним кількості вибірки з двох таблиць, тому, нехай M – розмір представлення. При цьому справедливий такий вираз:

$$M = N_1 + N_2,\ M \gg 100, \tag{3}$$

де M – обсяг представлення.

При цьому, згідно з публікацією [29], асимптотична складність вибірки у кращому випадку при використанні b-tree індексу становитиме $O(log_2(M))$. Незважаючи на високу швидкість отримання даних, інформація може бути потенційно застарілою, оскільки для оновлення інформації необхідно оновити саме представлення, що викличе повторне виконання часозатратного запиту, тому при постійному оновленні введення представлення у систему ніяк не змінить виявлених проблем складних запитів.

Розглянувши типові запити зміни даних – вставка, оновлення, видалення, та проаналізувавши частоту їх появи, було встановлено, що найбільш популярним

після вибірки є запит вставки. Враховуючи усі вищенаведені фактори, для кешування даних, які користувач створив у системі після оновлення матеріалізоване представлення було створено додаткові таблиці CT_I1 та CT_I2, що кешуватимуть дані, які будуть застосовані у вихідні таблиці шляхом виконання вставки, а саме представлення буде оновлено.

Нехай I_1 та I_2 – обсяги додаткових таблиць CT_I1 та CT_I2 відповідно. Після переміщення даних і оновлення представлення, додаткові таблиці будуть очищені, тому їх розмір буде відносно невеликим, тому вони становитимуть $I_1 \ll N_1$, $I_2 \ll N_2$, $N \gg 100$. На рисунку 2.12 подано оновлену схему із використанням допоміжних таблиць.

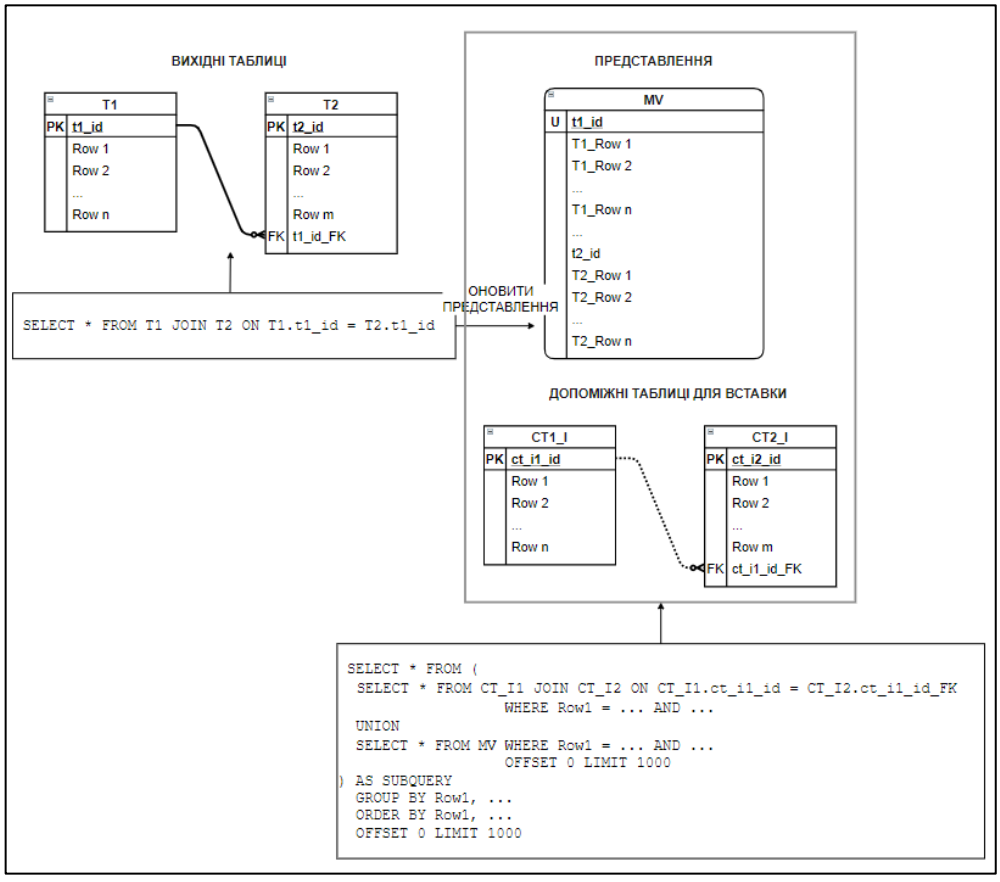


Рисунок 2.12 – Система із допоміжними таблицями для вставки

Таким чином вибірка з представлення продовжується постійно, а допоміжні таблиці для вставки забезпечують актуальність даних. При цьому асимптотична складність вибірки буде такою:

$$IT = O(log_2(M) + I_1 + I_2), \quad (4)$$

де IT – асимптотична складність алгоритму вибірки із представлення та допоміжних таблиць для вставки;

I_1, I_2 – кількість записів у першій та другій допоміжній кешуючій таблиці для вставки відповідно.

Якщо користувач надсилав запит на видалення даних, то виникає проблема, аналогічна до тієї, яка мала місце при вставці, а саме, проблема з актуальністю даних. Отже, для її вирішення необхідно створити подібні допоміжні таблиці, відмінністю яких буде лише те, що вони при застосуванні змін до вихідних таблиць видалятимуть дані. Тому було створено додаткові таблиці CT_D1 та CT_D2 із обсягами даних D_1 та D_2 відповідно. Їх розмір також невеликий, а тому: $D_1 \ll N_1$, $D_2 \ll N_2$, $N \gg 100$. Схема подана на рисунку 2.13.

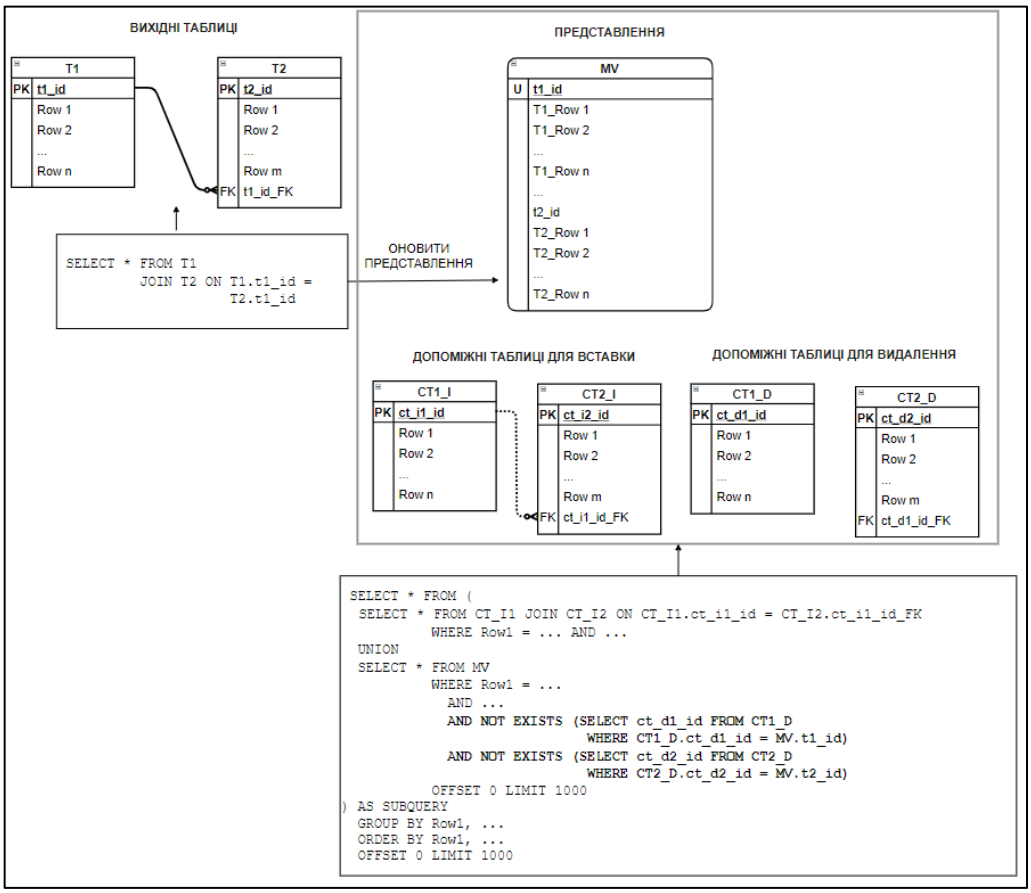


Рисунок 2.13 – Схема з використанням допоміжних таблиць для видалення

Як видно, для уникнення вибірки видалених даних застосовується оператор NOT EXISTS. Це забезпечує більш високу продуктивність ніж використання об'єднань типу left join згідно проведених тестів у [30]. Асимптотична складність буде такою:

$$IDT = IT + O(D_1 + D_2), \tag{5}$$

де IDT – асимптотична складність алгоритму вибірки із представлення та допоміжних таблиць для вставки і видалення;

D_1, D_2 – кількість записів у першій та другій допоміжній кешуючій таблиці для видалення відповідно.

Для виявлення видалених записів, тобто тих, які користувач вирішив видалити, проте які ще не застосувались у матеріалізованому представленні, використовуються під-запити. Швидкість виконання цих операцій залежить в першу чергу від обсягів допоміжних таблиць та від того, які індекси там встановлено. Якщо, наприклад, застосовано b-tree індекси на ключових полях, то вибірка буде відбуватись швидко, а тому сканування всієї таблиці виконуватись не буде, і під-запити будуть виконуватись швидше. Тобто, за наявності b-tree індексів можна окреслити нову асимптотичну оцінку, що характеризуватиметься такою складністю алгоритму:

$$IDT = IT + O(log_2(D_1 \cdot D_2)) \tag{6}$$

Таким чином, логарифмічна складність означає пришвидшення роботи під-запитів, що суттєво вплине на показники продуктивності усієї вибірки.

Ще одну операцію, яку варто розглянути – оновлення даних. Відносно інших вона є комплексною у реалізації. Для збереження даних для оновлення будуть використовуватись таблиці CT_U1 та CT_U2 із обсягами даних U_1 та U_2 відповідно. Причому : $U_1 \ll N_1, U_2 \ll N_2, N \gg 100$. Схема подана на рисунку 2.14.

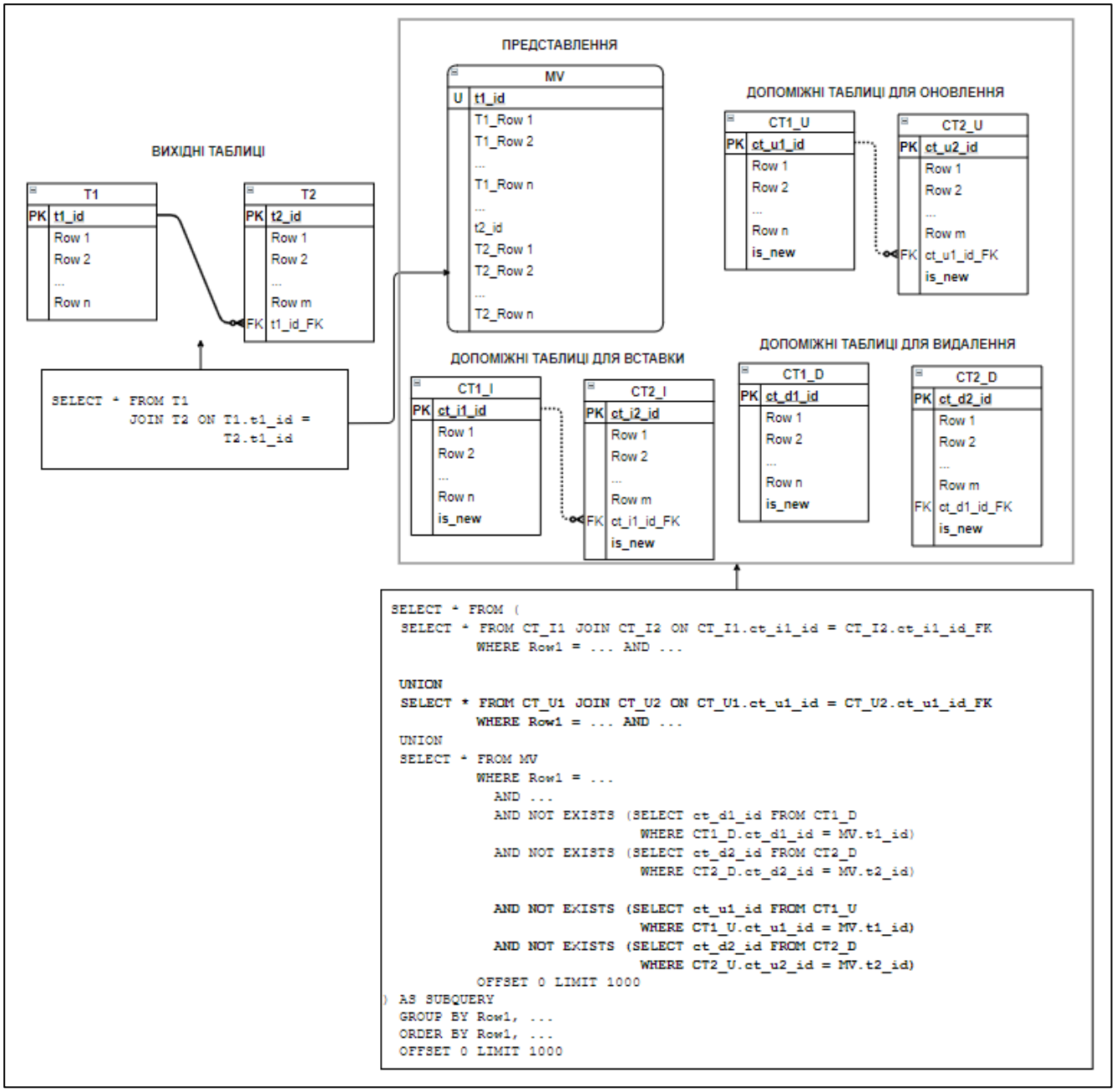


Рисунок 2.14 – Схема удосконаленого методу матеріалізованих представлень

Як видно з рисунку, спочатку виконується об’єднання даних із таблиці для оновлення із таблицею для вставки, а потім відфільтровуються у представлені потенційно оновлені та видалені дані, що забезпечує чистоту вибірки. Крім того додана допоміжна колонка is_new, що відділяє дані, змінені користувачем від тих, що було вибрано по ключу. Тому, ті, що змінив користувач будуть застосовуватись до вихідних таблиць, а вибрані дані – ні, оскільки вони уже існують.

Підсумовуючи, можна виділити загальну асимптотичну оцінку у кращому випадку для вибірки із системи, зображеної на рисунку 2.14:

$$\begin{gathered} B = IDT + O(U_1 + U_2 + log_2(U_1 \cdot U_2)) = \\ = O\big(log_2(M) + I_1 + I_2 + log_2(D_1 \cdot D_2) + U_1 + U_2 + log_2(U_1 \cdot U_2)\big) = \\ = O(log_2(M \cdot D_1 \cdot D_2 \cdot U_1 \cdot U_2) + I_1 + I_2 + U_1 + U_2) \end{gathered} \quad (7)$$

де B – асимптотична складність алгоритму вибірки із представлення та допоміжних таблиць для вставки, видалення та оновлення;

U_1, U_2 – кількість записів у першій та другій допоміжній таблиці для оновлення відповідно.

Слід зазначити, що наведені у (7) асимптотичні оцінки є приблизними, оскільки складність роботи алгоритму залежить від дій та алгоритмів оптимізатора конкретної СКБД, а тому це залежить від типу СКБД та налаштувань. Також у формулі (7) подано асимптотичну складність алгоритму у кращому випадку.

Якщо у системі є довільна кількість представлень та таблиць, асимптотична оцінка у кращому випадку становитиме:

$$B_t = O(log_2(\prod_{i=1}^{n} D_i \cdot \prod_{k=1}^{n} U_k \cdot M) + \sum_{j=1}^{n} I_j + \sum_{k=1}^{n} U_k), \quad (8)$$

де B_t – загальна асимптотична складність алгоритму вибірки при удосконаленому методі матеріалізованих представлень,

j, k, i – кількість даних у допоміжних таблицях для вставки, оновлення та видалення відповідно.

Якщо система не потребує видалення даних одразу, то у таких випадках дані у таблиці помічаються як видалені, проте не видаляються одразу, а значить відбувається їх оновлення. Тому часто допоміжні таблиці на видалення не потрібні, оскільки процес видалення і оновлення – це одне й те саме у таких випадках. З

урахуванням довільної кількості представлень, узагальнена схема роботи удосконаленого методу представлена на рисунку 2.15.

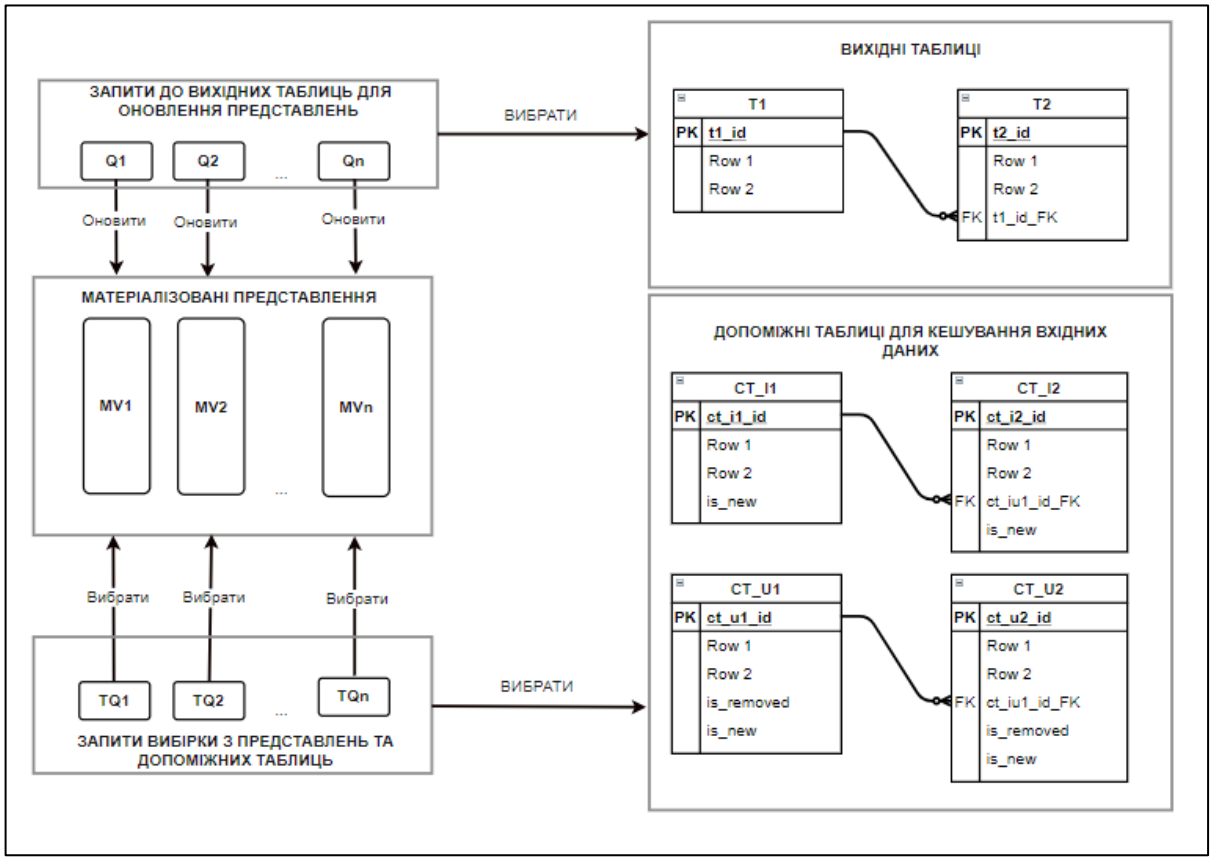


Рисунок 2.15 – Узагальнена схема роботи удосконаленого метод матеріалізованих представлень

Таким чином, удосконалений метод дозволяє використовувати матеріалізовані представлення на постійній основі у поєднання із додатковим шаром кешування у вигляді допоміжних таблиць. Швидкість вибірки залежить від кількості даних у допоміжних таблицях, а також від їх кількості та від того, чи була проведена їх індексація.

2.3 Висновки

У розділі здійснено аналіз існуючих методів та підходів до вирішення задачі реінжинірингу. Наприклад, класичні методи можна розподілити на етап редизайну та декомпозиції. При цьому якщо наявні методи редизайну не допомагають вирішити наявні проблеми з продуктивність, це означає, що або даних занадто велика кількість або частина даних має невластиву обраній системі керування базами даних структурі. Тому декомпозиція та міграції може бути застосована у таких випадках та оптимально вирішити проблеми з продуктивністю шляхом повної або часткової міграції даних у інший тип сховища, або використовувати мульти-модельні бази даних, що представляють собою готові програмні рішення, які дозволяють обробляти та містити дані різної структури. Було встановлено, що окрім переваг, наявні методи реінжинірингу бази даних мають і суттєві недоліки. Наприклад, якщо лише невелика частина даних зберігається у невластивій структурі, то декомпозиція сховища може займати велику кількість часу, а фінансові витрати на додаткові апаратно-програмні засоби будуть відчутними.

Ці проблеми запропоновано вирішити за допомогою удосконалення методу матеріалізованих представлень. Було досліджено швидкість роботи, встановлено асимптотичні оцінки та доцільність використання матеріалізованих представлень при вибірці даних. Також було з’ясовано, що представлення можна використовувати і на постійній основі, застосувавши шар кешування, представлений додатковими таблицями, які містять дані на вставку, видалення та оновлення. Так як допоміжні таблиці будуть очищатись по мірі оновлення представлень, швидкість вибірки із масивних таблиць суттєво зросте.

На основі удосконаленого методу було розроблено моделі, які представляють собою ER-діаграми рішень та наведено гіпотетичні асимптотичні оцінки удосконаленого методу. Наступним етапом роботи є проектування та декомпозиція програмного рішення на основі розроблених моделей, а також підбір та обґрунтування засобів реалізації.

3 ТЕХНОЛОГІЯ РЕАЛІЗАЦІЇ УДОСКОНАЛЕНОГО МЕТОДУ МАТЕРІАЛІЗОВАНИХ ПРЕДСТАВЛЕНЬ

3.1 Аналіз вимог до програмного засобу

На основі аналізу предметної області, розробленого методу та алгоритмів, необхідно сформувати та описати вимоги до створюваного програмного засобу, метою розробки якого є імплементація удосконаленого методу матеріалізованих представлень у реінжинірингу бази даних. Програмним засобом виступатиме бібліотека, яку розробники зможуть застосувати при створенні власного програмного забезпечення. Можна виділити такі характеристики бібліотеки:

– автоматичне створення допоміжних таблиць до вихідних, що помічені спеціальними атрибутами;

– автоматичне створення матеріалізованих представлень на основі спеціальних дата-моделей на рівні програмного коду;

– застосування спеціального запиту, розглянутого у розділі 2.2, при вибірці із матеріалізованих представлень;

– виконання алгоритму застосування змін та створення автоматичного планувальника завдань, який визначатиме терміни оновлення представлення;

– можливість застосування до різних СКБД, що підтримують концепцію матеріалізації даних.

1) Класи користувачів та їх характеристики

Система матиме лише один клас користувачів – розробників програмного забезпечення. Користувачі зможуть виконувати лише вхідні запити, решта роботи виконуватиметься автоматично на основі алгоритмів.

2) Середовище функціонування

Середовищем функціонуванням є додаток, який побудований на основі клієнт-серверної архітектури; при цьому розроблюваний програмний засіб має бути незалежним модулем (бібліотекою) та мати можливість підключатись у проект як інтегрований плагін.

3) Характеристики системи

Система повинна автоматично забезпечувати створення додаткових таблиць та матеріалізованого представлення лише один раз при першому запуску додатку. Вибірка даних із матеріалізованого представлення повинна відбуватись з урахуванням удосконаленого методу та із застосуванням алгоритму, приведеному у розділі 2.2. Також повинен створюватись планувальник завдань, який буде у визначений період часу очищати допоміжні таблиці та оновлювати представлення. Усі записи про успішні операції та помилки повинні записуватись у логи. Вставка, оновлення та видалення повинні автоматично записуватись як в оригінальні, так і в допоміжні таблиці для забезпечення цілісності даних. Для бібліотеки повинні бути написані модульні тести.

3.2 Проектування програмного засобу

Розроблюваний програмний засіб має працювати як окремий плагін, тобто буде складовою будь-якої ПС. Таким чином, розробники будуть взаємодіяти з ним не напряму, а опосередковано.

Для проектування засобу було використано парадигму об’єктно-орієнтованого програмування. В загальному метою бібліотеки є автоматизація процесів та алгоритмів, описаних та спроектованих у розділі 2.2. Відносно запиту, за допомогою якого побудоване представлення можна аналізувати його частини, виконати розбір, та автоматично визначити умови та таблиці, які необхідно буде застосувати при вибірці даних із використанням удосконаленого методу матеріалізованих представлень. Така архітектура значно пришвидшить розробку програмних систем із удосконаленим методом та вимагатиме від розробників меншого зосередження на рутинному процесі створення додаткових таблиць для кешування даних та матеріалізованих представлень. На рисунку 3.1 продемонстровано узагальнену діаграму класів бібліотеки.

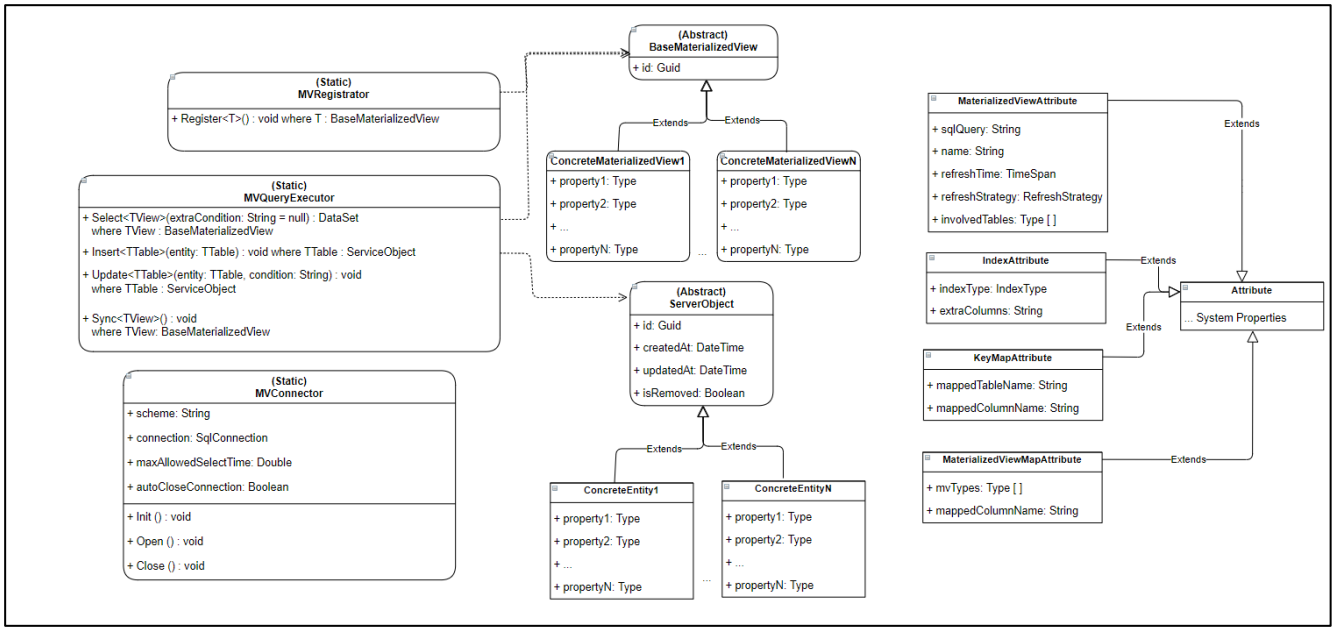


Рисунок 3.1 – Діаграма класів реалізації удосконаленого методу матеріалізованих представлень

Проаналізуємо детально компонентну структуру. Реєстрація матеріалізованих представлень та допоміжних відбуватиметься одразу на початку виконання програми за допомогою класу MVRegistrator. Вона включатиме в себе створення представлень та кешуючих таблиць, якщо таких ще немає, а також записувати їх у так званий реєстратор, що на основі часу оновлення кожного представлення (поле refreshTime в класах, що представляють об’єктні моделі представлень) та запускає фоновий таймер, який буде перевіряти час, чи потрібно виконати оновлення. Виконання оновлення відбуватиметься у методі Sync класу MVQueryExecutor.

Абстрактний клас ServerObject – це звичайний дата клас, який містить усі базові поля та властивості для усіх моделей у одній схемі бази даних. А конкретні реалізації класу можуть містити свої особисті поля та властивості (на рис. 3.1 – ConcreteEntity1, …, ConcreteEntityN.

Абстрактний клас BaseMaterializedView виступає базовим для усіх матеріалізованих представлень та містить загальні поля та властивості, такі як: id – унікальний ідентифікатор. Йому відповідає атрибут MaterializedView, що застосовується до класів та містить такі поля, як: refreshTime – час оновлення

представлення, sqlQuery – запит у вигляді строки, за допомогою якого було побудовано представлення, involvedTables – масив типів, які беруть участь у формуванні додаткових кешуючих таблиць. За допомогою цього поля система буде «знати», як виконувати запит із використанням допоміжних таблиць, та які допоміжні таблиці необхідно створити при запуску програми (якщо їх немає).

Клас MVQueryExecutor виконує основну роботу – запити відповідно спроектованим алгоритмам та моделям. Запит Select вибирає дані із представлення та допоміжних таблиць, Insert – створює записи у вихідних та допоміжних таблицях, а Update – оновлює їх із використанням предикату. Sync виконує синхронізацію представлення та допоміжних і вихідних таблиць.

Клас MVConnector має функціонал ініціалізації і керування підключенням до серверу бази даних там містить усі необхідні методи та поля.

Також допоміжні класи-атрибути реалізовують парадигму мета-програмування, тобто надають допоміжну інформацію про поля та класи. Наприклад, за допомогою атрибуту IndexAttribute, можна повідомити систему про те, що при створенні представлення необхідно і створити індекс на вказаному полі із визначеним типом. KeyMapAttribue описуватиме поля, які до потрапляння у представлення були ключовими полями у кожній зв’язаній таблиці. Таким чином при вибірці буде зрозуміло, як назвати поля та як з ними взаємодіяти алгоритму.

Розглянемо детально алгоритм вставки при використанні удосконаленого методу матеріалізованих представлень. Якщо користувач захоче створити новий запис у таблиці, що мають образи у вигляді допоміжних кешуючих таблиць та застосовуються у алгоритмі вибірки із пов’язаного матеріалізованого представлення, то вставки у основну таблицю буде недостатньо. Необхідно також виконати створення даних у відповідній допоміжній таблиці. А якщо таблиця має ще зв’язки з іншими, то простої вставки буде замало. Для забезпечення консистентності даних і допоміжну зв’язану таблицю необхідно записати дані, що є в оригінальній, тобто скопіювати їх та закешувати у допоміжній таблиці. Таким чином, при вибірці дані будуть правильно зчитані і не виникне необхідності виконувати вибірку із оригінальних таблиць при запитів читання з бази даних.

Проте, якщо є така необхідність або даних у зв’язаних таблицях не може бути невизначена кількість, то запит можна виконувати і без необхідності створення додаткової таблиці для зв’язаної.

Також, якщо дотримуватись методу пост-оновлення для пришвидшення процесу вставки, можна виконувати перенесення усіх нових даних із допоміжної таблиці до оригінальної уже на етапі виконання оновлення матеріалізованого представлення. Таким чином перевага такого способу є швидша вставка, проте недоліком є довше оновлення представлення, що може вплинути негативно на продуктивність сховища даних під час процесу оновлення. Діаграму діяльності проілюстровано на рисунку 3.2.

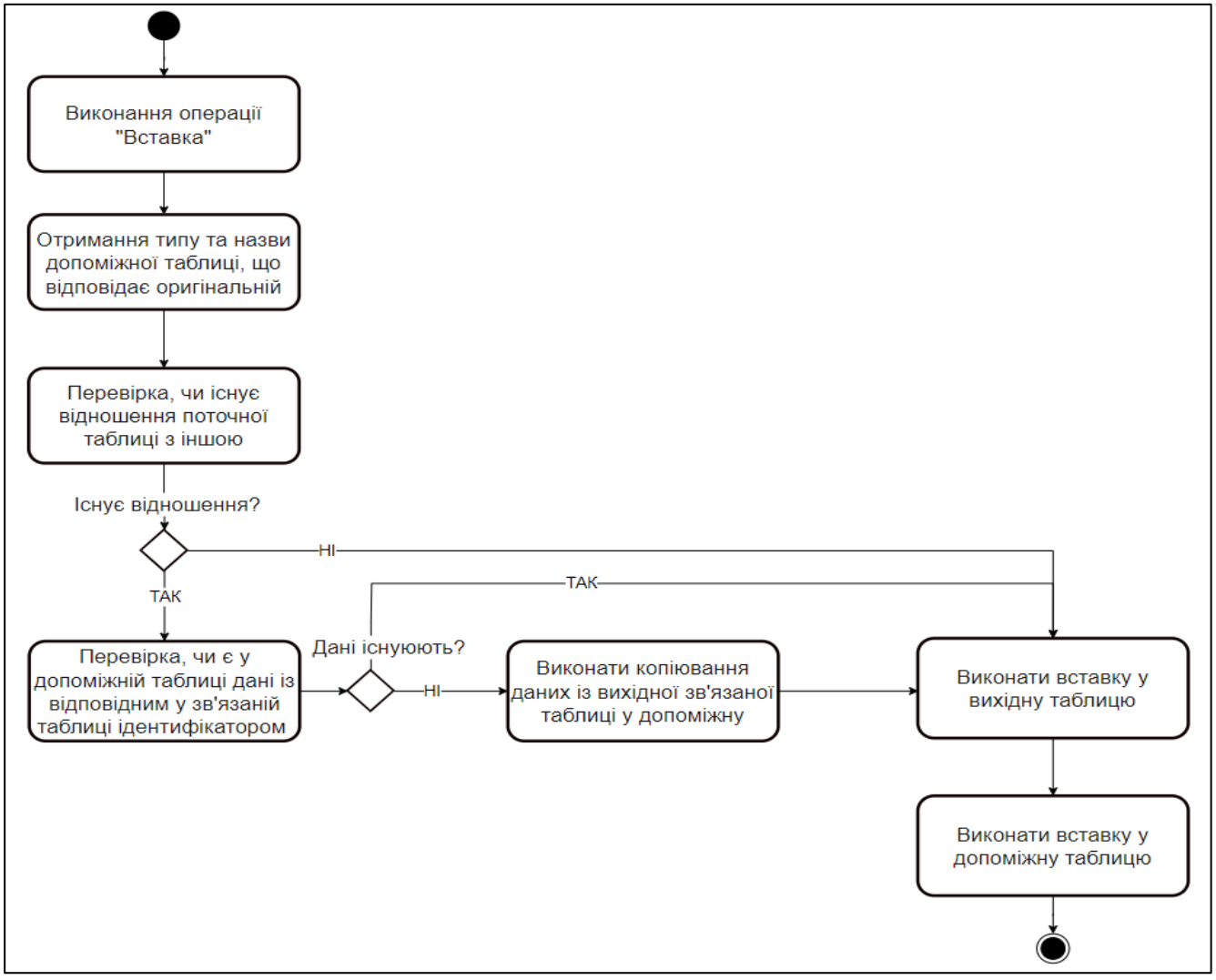


Рисунок 3.2 – Алгоритм вставки при використанні удосконаленого методу матеріалізованих представлень

Слід зазначити, що копіювання даних із вхідної таблиці у допоміжну, вставку в вихідну і таблицю кешування повинні відбуватись у одній транзакції, оскільки при виникненні помилки на одному з етапів може відбутись порушення цілісності даних, а транзакція гарантує або повне успішне виконання, або відновлення попереднього стану даних у разі виникнення непередбачуваних помилок. Узагальнений лістинг запиту буде таким:

```
BEGIN TRANSACTION;

DECLARE
@exists_in_T1 boolean,
…,
@exists_in_TN boolean;
SELECT @exists_in_T1=EXISTS(SELECT t1_id FROM T1 WHERE t1_id = @rel_t1_id;
…
SELECT @exists_in_TN=EXISTS(SELECT tn_id FROM TN WHERE tn_id = @rel_tn_id;
IF @exists_in_T1 = FALSE
      INSERT INTO CT_I1 (ct_i1_id, …, is_new)
      SELECT t1_id, …, FALSE FROM T1 WHERE t1_id = @rel_t1_id;
…
INSERT INTO T (t_col1, …) VALUES (@t_val1, …)
INSERT INTO CT_I (ct_i_col1, …) VALUES (@t_val1, …);

COMMIT;
```

У запиті T, T1, …, TN – вихідні таблиці, CT_I, CT_I1, … , CT_IN – допоміжні. Запит демонструє приблизну механіку алгоритму вставки.

Розглянемо процес оновлення. Він майже аналогічний вставці даних, оскільки так само використовує як допоміжну, так і оригінальну таблиці. Проте незмінним залишається лише вибірка даних зі зв’язаних таблиць. Так само необхідно скопіювати дані у допоміжні таблиці для встановлення сумісності на рівні кешування та застосовувати оновлення усередині транзакцій.

Процес видалення може не застосовуватись взагалі, оскільки для багатьох систем видалення відбувається у фоновому режимі одразу багатьох записів із відповідною поміткою, яка вказує, що сутність потрібно видалити. Тому видалення у більшості систем це звичайне оновлення, механізм якого наведено вище. Якщо ж операцію видалення розглядати окремо, то механізм роботи буде аналогічний двом попередньо розглянутим.

Механізм оновлення представлення в удосконаленому методі має свої особливості, оскільки, крім власне оновлення необхідно очищати допоміжні таблиці. З першого погляду це неважке завдання, однак, необхідно встановити так звані контрольні точки – помітки у часі, коли представлення оновилось, чи взагалі оновилось, коли очистились допоміжні таблиці та чи усі дані видалено. Для цього можна застосувати окремі таблиці – мета-таблиці, тобто ті, що містять службову інформацію про інші об’єкти бази даних. Крім того, процес видалення часто не є швидким та блокує таблиці, що унеможливлює їх використання під час видалення. Це серйозна перешкода у ефективному оновленні. Є декілька варіантів рішень.

Одна із опцій – створення додаткової таблиці до допоміжної, яка буде використовуватись тільки при вибірці даних. Таким чином можна використати метод, що називається свапінгом, суть якого у тому, щоб перед оновленням представлення та видаленням даних із допоміжних таблиць, копіювати весь вміст останніх та переміщувати у додаткові таблиці для свапінгу. При цьому у мета-таблицях позначати після завершення копіювання, щоб на час видалення, робота продовжувалась із додатковою таблицею, а після видаленні із основної допоміжної таблиці за тригером у мета-таблиці вказуватиме на неї і робота уже буде із основною допоміжною таблицею.

Інший варіант полягає у повністю видаленні та створенні таблиці заново. При цьому знову ж таки дані мають бути скопійовані у додаткову таблицю. Цей варіант може бути швидшим за попередній, оскільки видалення таблиці є значно швидшим процесом, ніж видалення даних з таблиць, адже кожне видалення записується у логи та сповільнює сам процес.

Проте, якщо даних у допоміжних таблицях небагато, то можна використовувати стандартний процес видалення без застосування додаткових дій. Проте необхідно знати, чи оновилось матеріалізоване представлення, оскільки видалення даних відбуватиметься лише після оновлення представлення. Для цього можна у запиті, результат якого необхідно зберегти, вказати допоміжну колонку із датою створення запису. Таким чином раз у декілька хвилин можна перевіряти, чи не з’явилися дані у представленні, дата яких пізніша за попередньо оновлену, яку

було занесено у мета-таблицю. Саме ж представлення варто оновлювати із використанням команди concurrent, що означає, що блокування не буде, проте продуктивність вибірки може трохи знизитись за рахунок навантаження на представлення та конкурентне оновлення. Але саме оновлення – не часте явище, а тому у загальному на продуктивність не вплине. Алгоритм подано на рисунку 3.3.

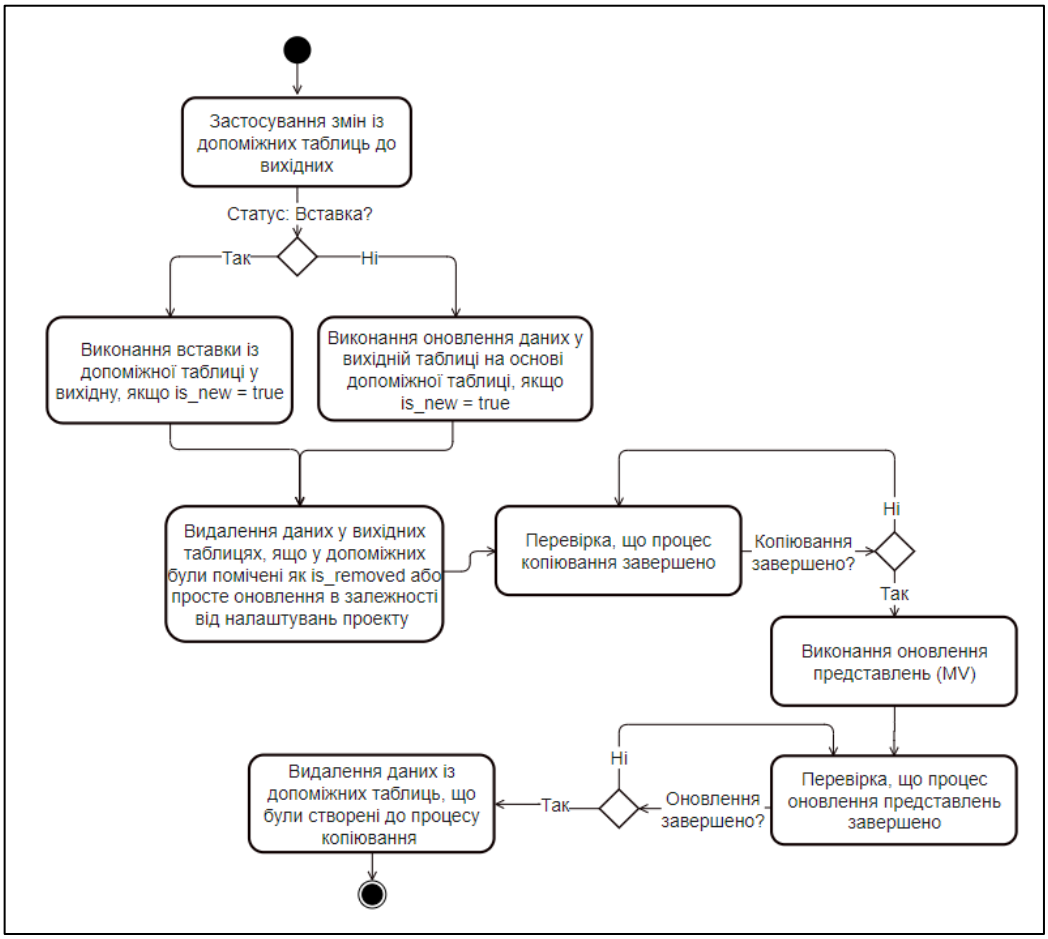


Рисунок 3.3 – Алгоритм синхронізації допоміжних та вихідних таблиць

Як видно з алгоритму, для застосування змін потрібен певний час, а тому важливо коректно обчислювати період невеликого навантаження на сервер. Є декілька способів встановлення періоду синхронізації. Перший залежить від кількості даних у допоміжних таблицях. Тобто, спочатку вираховується розмір таблиць, при яких продуктивність почне деградувати. Для цього можна скористатись асимптотичною оцінкою, що подана у розділі 2.2 (8). Це означає, що аналітично можливо визначити розмір допоміжних таблиць, враховуючи їх

кількість та кількість даних у них та матеріалізованому представленні та підставивши у 2.2 (8) усі необхідні значення змінних. Проте для цього необхідно завжди виконувати запити обрахування кількості даних, що може вплинути на продуктивність. Можливо денормалізувати лічильники шляхом запису їх у мета-таблицю, проте на саму продуктивність впливає багато інших чинників, окрім кількості таблиць та їх обсягу. Це залежить від конкретного оптимізатора конкретної системи керування базами даних. Тому цей спосіб є не дуже дієвим для обрахування часу оновлення представлення.

Інший спосіб полягає у завчасному виконанні плану запиту вибірки в удосконаленому методі матеріалізованих представлень. Таким чином, разом із запитом вибірки посилатиметься запит типу Explain-Analyze, за результатами якого можна дізнатися приблизний час виконання запиту. При цьому необхідно написати спеціальну функцію, яка б виконувала парсинг цих результатів і на основі них визначала б, чи необхідно оновлювати представлення та синхронізувати зміни. При цьому, можна не використовувати фіксовані проміжки часу оновлення, оскільки рішення про синхронізацією прийматиметься не на основі запланованого завдання або таймеру, а на основі часу виконання запиту. Недоліком може бути, що неможливо одночасно виконати два запити, а їх послідовність негативно вплине на продуктивність. Проблему можна вирішити, якщо замість запиту Explain використати функції вимірювання тривалості виконання методу у програмному коді наявними інструментами мови програмування. Таким чином, запит буде посилатись лише один та зникне необхідність у написанні додаткової функції для парсингу плану запиту.

Ще один метод вирахування періоду оновлення – це стандартне задання фіксованого часу, коли необхідно синхронізувати зміни (наприклад, один раз на день, тиждень тощо). Проте це може бути неефективним рішенням, оскільки дата оновлення може припасти випадково на найвище навантаження серверу. А тому даний метод використовують із вирахуванням оптимального часу на основі показників завантаженості серверу, кількості активних користувачів тощо. На основі показників вибиратиметься майбутній час оновлення у заданих часових

рамках. Можливе використання елементів машинного навчання для побудови нейронної мережі, що прогнозуватиме високе навантаження та визначатиме оптимальну дату або проміжок часу для оновлення. Недоліком є складність реалізації методу, оскільки нейронні мережі мають певний час навчатися на готовій системі, а отже, у цей період, поки мережа недосконала, будуть застосовуватись фіксовані проміжки часу, що певним чином вплинуть на загальну продуктивність.

Тому у роботі запропоновано використовувати метод оновлення матеріалізованих представлень та синхронізації змін допоміжних і вихідних таблиць, який полягає у тому, що система виконуватиме розрахунок часу виконання запиту. При цьому, якщо результат буде постійно вищим допустимої константи (за високу продуктивність часто беруть час виконання запиту, який є менше однієї секунди), то обрахунок виконуватиметься лише один раз у певний визначений проміжок часу, щоб не допустити частого оновлення представлень. Якщо результат частіше є меншим допустимої константи, то оновлення і синхронізація відбуватиметься як завжди, тобто при досягнення максимально допустимого значення.

Якщо ж результат обрахунку продуктивності завжди є недопустимим, це означає що система побудована неправильно, або присутнє невірне використання індексів на полях, що підлягають фільтрації, пошуку або сортуванню.

На рисунку 3.4 подано алгоритм, який показує реалізацію вирахування необхідності оновлення та синхронізації за наведеним вище методом.

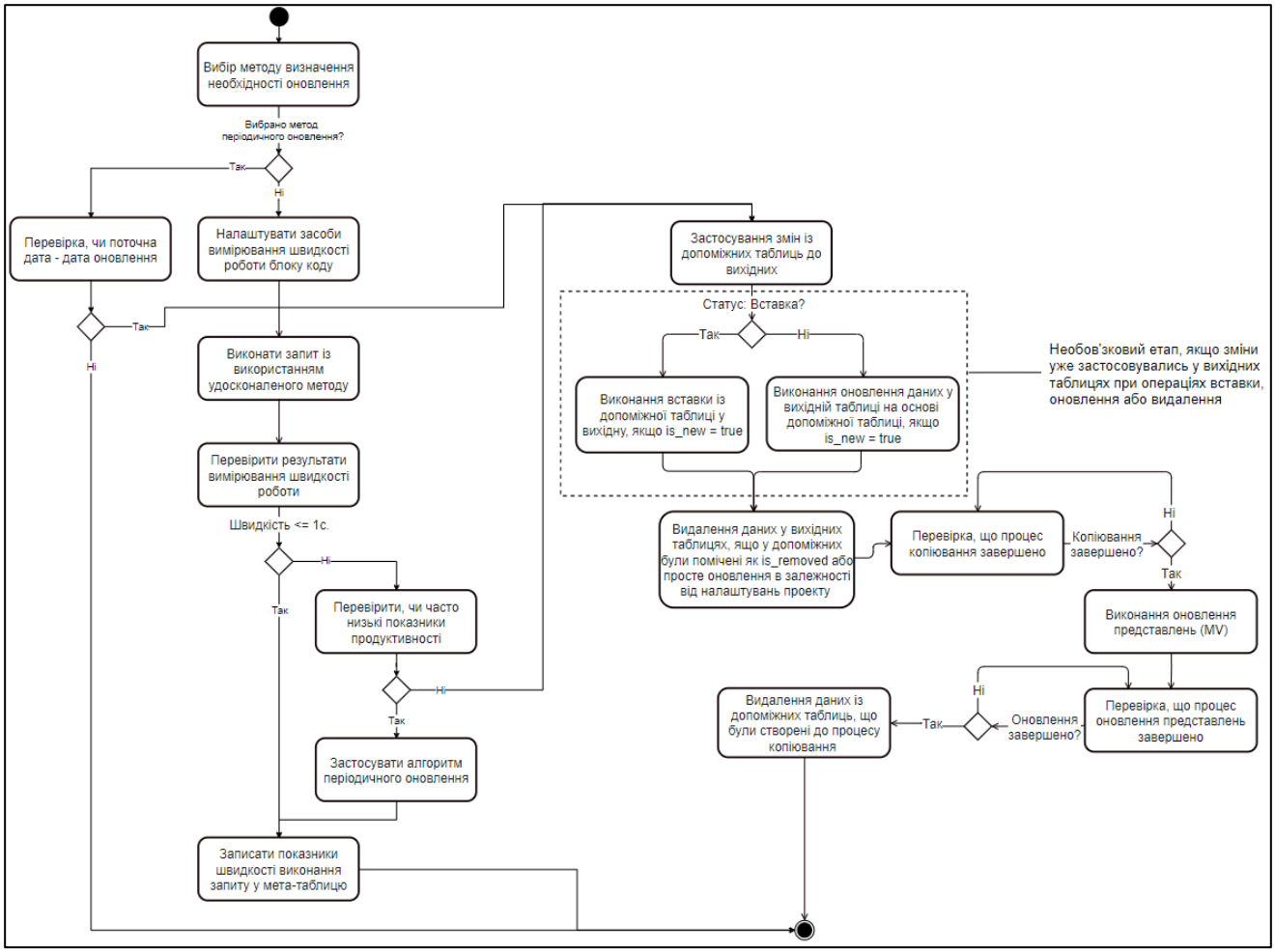


Рисунок 3.4 – Алгоритм визначення необхідності оновлення та синхронізація змін

3.3 Аналіз та вибір засобів програмної реалізації методу

Під час розробки ПЗ слід врахувати додаткові вимоги: засіб має бути гнучким, а саме: легко розширюватись та інтегруватись у будь-яке середовище. З огляду на це, було вирішено використати мову програмування C# та платформу .NET – безкоштовне кероване програмне забезпечення з відкритим кодом для операційних систем Windows, Linux і macOS. Це кросплатформний наступник .NET Framework, що дозволяє запускати програми у мультиплатформеному режимі. Крім того, бібліотека буде незалежна від фреймворку, тобто її можна використовувати у будь-яких додатках, що написані із використанням платформи.

Крім того, сама бібліотека міститиме плагін для взаємодії з базою даних Microsoft.Data.SqlClient. Оскільки не можна визначити конкретну систему для

застосування бібліотеки, автономний плагін не потребує окремих рішень, таких як ін’єкція залежностей, що реалізовано у ASP.NET.

Крім того, для вимірювання часу виконання методу було використано плагін System.Diagnostics.Stopwatch для визначення, чи необхідно оновлювати матеріалізоване представлення та синхронізувати допоміжні та вихідні таблиці.

У якості бази даних було обрано PostgreSQL, оскільки вона є однією із найпопулярніших систем керування, що застосовуються у розробці корпоративних онлайн додатків. Проте, є можливість розширювати бібліотеку, додаючи нові провайдери баз даних, які підтримують концепцію матеріалізації даних.

Також було використано бібліотеку XUnit. xUnit.net – це безкоштовний інструмент модульного тестування з відкритим вихідним кодом, орієнтований на використання у .NET Framework, проте також є підтримка для .NET 6.

3.4 Висновки

У розділі проведено аналіз вимог до програмного засобу – бібліотеки, що реалізуватиме концепції та моделі удосконаленого методу матеріалізованих представлень. Було розроблено діаграму класів бібліотеки, які включають функціонал реєстрації плагіну, автоматичне створення представлень із допоміжними кешуючими таблицями, а також реалізація основних концепцій та алгоритмів удосконаленого методу матеріалізованих представлень. Крім того розроблено діаграми діяльності для вставки даних у вихідні та допоміжні таблиці, а також, для синхронізації даних між допоміжними та вихідними таблицями. Також обрано і спроектовано алгоритм оптимального підбору часу оновлення представлень із метою максимального зниження навантаження на сервер та підвищення продуктивності виконання запитів вибірки. Обрано та обґрунтовано перелік технологій, які є необхідними для вирішення задачі. Наступним етапом є реалізація спроектованого ПЗ та його тестування, а також доведення ефективності удосконаленого методу матеріалізованих представлень.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАСОБУ

4.1 Програмна реалізація

Для того, щоб розробникам було легше використовувати узагальнений метод матеріалізованих представлень у розробці програмних систем, було вирішено створити бібліотеку Advanced Materialization Library, яка забезпечить потрібним функціоналом та автоматизує процеси створення представлень, кешуючих таблиць, їх налаштування та виконання вибірки і оновлення. Для цього було створено окремий проект бібліотеки, який має просту структуру, що складається з трьох модулів: Attributes, Logic, Internal.

Модуль Attributes представляє реалізацію класів-атрибутів, які відповідають за надання додаткової інформації, що необхідна при виконанні вибірки, створення представлень, кешуючих таблиць тощо. Тобто основний принцип, який використовується при роботі з атрибутами – це рефлексія, тобто виконання операцій над семантичними конструкціями програмного коду, як-от: отримання властивостей класу та його атрибутів, отримання аргументів конструктору атрибутів властивості тощо.

Перший і основний атрибут у системі – це MaterializedViewAttribute. Він представляє собою клас, що надає інформацію про те, як у подальшому обробляти класи, що помічені даним атрибутом, а саме, які налаштування необхідно застосувати під час створення матеріалізованих представлень та допоміжних таблиць. Лістинг коду подано нижче:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class MaterializedViewAttribute : Attribute
{
  public string Name { get; init; }
  public TimeSpan RefreshTime { get; init; } = new(1, 0, 0, 0);
  public RefreshStrategy RefreshStrategy { get; init; } =
         RefreshStrategy.Auto;
  public Type[] InvolvedTables { get; init; } = Array.Empty<Type>();
  public MaterializedViewAttribute(string sqlQuery)
  {
  }
}
```

Поле Name вказує на ім’я матеріалізованого представлення як у базі даних.

RefreshTime показує, який період часу необхідно застосовувати для оновлення представлення. При цьому він може використовуватись лише тоді, коли стратегія представлення не автоматична.

RefreshStrategy визначає стратегію, відповідно до якої буде відбуватись оновлення представлення. Є два типи – Auto та Scheduled. Перша означає алгоритм синхронізації, який полягає у вимірюванні часу виконання методу вибірки з представлення та допоміжних таблиць, та аналізі вимірювань і, якщо час виконання перевищує встановлену у налаштуваннях допустиму норму, то необхідно виконати оновлення представлення. Scheduled – стратегія може використовуватись, коли необхідно оновити представлення лише один раз у певний проміжок часу, що визначається у властивості RefreshTime.

InvolvedTables описує типи класів, які представляють таблиці, що використовуються у запитів в представленні. Таким чином використовуючи рефлексію, можна проаналізувати дані, що містять об’єкти і на основі них автоматизувати побудову складних запитів, де ці дані (поля, атрибути) важливі.

Аргумент конструктору sqlQuery – застосовується для зберігання запиту, на основі якого заповнюється матеріалізоване представлення.

Клас атрибуту IndexAttribute використовується як маркер, що позначає поля, на які необхідно встановити індекси при створенні матеріалізованих представлень та таблиць. При цьому він містить поле type, що вказує, який тип індексу необхідно створити. Існують декілька типів індексів у PostgreSQL і усі вони наведені у перерахуванні IndexType. Також є властивість ExtraColumns, що вказує на додаткові стовпці у базі даних, які необхідно влкючити до індексу. Лістинг коду атрибуту подано нижче:

```
[AttributeUsage(AttributeTargets.Property)]
public class IndexAttribute : Attribute
{
   public string ExtraColumns { get; set; }
   public IndexAttribute(IndexType type = IndexType.Btree) {}
}
```

KeyMapAttribute використовується, для того щоб встановити, які поля у класі, що представляють представлення будуть такими, відносно яких буде відбуватись комбінована вибірка. В основному це поля, які у своїх таблицях є ключовими, тобто це ідентифікатори. Атрибут має такий вигляд:

```
[AttributeUsage(AttributeTargets.Property)]
public class KeyMapAttribute : Attribute
{
   public KeyMapAttribute(string mappedTableName, string mappedColumnName)
   {
   }
}
```

Аргумент конструктору mappedTableName показує, до якої таблиці відноситься поле, а mappedColumnName – до якого стовпця у власній таблиці.

Модуль Logic виступає у якості основного, оскільки він реалізовує логіку удосконаленого методу. Там містяться основні класи, які виконують операції над матеріалізованими представленнями та таблицями.

MVConnector – клас, що використовується у якості вхідної точки у бібліотеку, тобто він ініціює і встановлює з’єднання із базою даних за допомогою методу Init, який приймає об’єкт з’єднання із бібліотеки ADO.NET. Вона використана проекті, оскільки бібліотека має бути універсальною і зручною у використанні не тільки у веб-сервісах, побудованих на ASP.NET, а й інших, зокрема і кросплатформених додатках. Також передається назва схеми, оскільки постгрес підтримує їх декілька і можна створювати власні схеми у одній базі даних. Аргумент maxAllowedSelectTime – налаштування, при якому система буде перевіряти допустимий час виконання запиту вибірки удосконаленим методом і, якщо він вище, ніж встановлено, то представлення буде оновлюватись. Тому не рекомендовано використовувати невеликі значення, оскільки представлення у такому разі буде оновлюватись частіше, що може негативно вплинути на продуктивність бази. Аргумент autoCloseConnection вказує, чи необхідно автоматично закривати з’єднання з базою даних. Це зроблено з метою забезпечення

закриття з’єднання у системах, які не використовують загально відомі ОРМ системи, що можуть самі керувати з’єднанням, та закривати його коли потрібно.

У класі також присутні методи Open і Close, що відкривають та закривають з’єднання відповідно. Вони доступні для використання лише всередині бібліотеки. Код виглядає таким чином:

```
public static class MVConnector
{
        public static string Scheme { get; private set; }
        public static SqlConnection Connection { get; private set; }
        public static double? MaxAllowedSelectTime { get; private set; }
        public static bool? AutoCloseConnection { get; private set; }
        public static void Init(
                                SqlConnection connection,
                                string scheme = "public",
                                double maxAllowedSelectTime = 1.0,
                                bool autoCloseConnection = false)
        {
            Connection ??= connection;
            Scheme ??= scheme;
            MaxAllowedSelectTime ??= maxAllowedSelectTime;
            AutoCloseConnection ??= autoCloseConnection;
        }

        internal static void Open()
        {
            if (Connection.State != ConnectionState.Open)
                Connection.Open();
        }

        internal static void Close()
        {
            if (AutoCloseConnection == true)
                Connection.Close();
        }
}
```

MVRegistrator виконує функцію створення представлень та зв’язаних із ним допоміжних кешуючих таблиць за допомогою методу Register, що приймає типізований аргумент, тип якого є будь-який клас, що містить атрибут MaterializedView. На основі даних класу, отриманих за допомогою використання механізму рефлексії, створюється представлення у базі даних, якщо такого немає. Потім на основі властивості атрибуту InvolvedTables, створюються допоміжні кешуючі таблиці із автоматично згенерованими назвами:

```
    public static class MVRegistrator
    {
        public static void Register<T>() where T : class
        {
            var type = typeof(T);
            var mvAttribute = MVHelper.GetMaterializedViewAttribute(type);
            var attributeData = MVHelper.ParseAttribute(mvAttribute, typeof(T));
            var connection = MVConnector.Connection;
            MVConnector.Open();
            string createMVSql =
                @$"CREATE MATERIALIZED VIEW IF NOT EXISTS
{MVConnector.Scheme}.{attributeData.Name} AS {attributeData.SqlQuery} WITH DATA";
            var command = new SqlCommand(createMVSql, connection);
            command.ExecuteNonQuery();
            MVHelper.CreateIndexes(type.GetProperties().ToList());
            if (attributeData.InvolvedTables.Count == 0)
                return;
            foreach (var t in attributeData.InvolvedTables)
            {
                var table = t.GetCustomAttributesData().FirstOrDefault(it =>
it.AttributeType == typeof(TableAttribute));
                var tableName = t != null ?
table.ConstructorArguments[0].Value.ToString() : t.Name;
                var foreignKeys = new List<string>();
                var props = t.GetProperties().Select(it =>
                {
                    var attrData = it.GetCustomAttributesData().FirstOrDefault(it
=> it.AttributeType == typeof(ColumnAttribute));
                    var attrForeignKey =
it.GetCustomAttributesData().FirstOrDefault(it => it.AttributeType ==
typeof(ForeignKeyAttribute));

                    if (attrForeignKey != null &&
attributeData.InvolvedTables.Contains(it.PropertyType))

foreignKeys.Add(attrForeignKey.ConstructorArguments[0].Value.ToString());
                    return it;
                }).ToList();

            MVHelper.CreateTable($"ct_i_{tableName}", props, foreignKeys);
            MVHelper.CreateTable($"ct_u_{tableName}", props);
            }
            MVConnector.Close();
        }
    }
```

Як правило реєстратори використовують при створенні моделі у випадку Code-First підходу, відповідно до методології OPM бібліотеки EntityFramwork. Тому метод Register може викликатись у методі OnModelCreating. У інших випадках, реєструвати необхідно одразу після ініціалізації.

Наступний клас – найважливіший – MVQueryExecutor, що виконуэ основны операції над представленнями та допоміжним таблицями. Він містить чотири методи: Select, Insert, Update та Sync.

Select відповідає за вибірку даних удосконаленим методом матеріалізованих представлень. Якщо не створено допоміжних таблиць, то вибірка відбувається звичайним чином, використовуючи лише матеріалізоване представлення. У іншому випадку, використовується удосконалений у розділі 2.2 метод. Код методу виглядає так:

```
public static DataSet MVSelect<TView>(string whereCondition = null, int offset =
0, int limit = 1000, string extraOperations = null) where TView : class
        {
            var type = typeof(TView);

            var mvAttribute = MVHelper.GetMaterializedViewAttribute(type);
            var mvData = MVHelper.ParseAttribute(mvAttribute, type);

            string selectFromInsertSql = null;
            string selectFromUpdateSql = null;
            string existsSubQueries = null;
            string sql = $"SELECT * FROM {mvData.Name} WHERE {whereCondition}
{extraOperations} OFFSET {offset} LIMIT {limit}";
            var involvedTablesNames = new List<string>();

            if (mvData.InvolvedTables.Count != 0)
            {
                involvedTablesNames = mvData.InvolvedTables.Select(it =>
                {
                    var table = it.GetCustomAttributesData().FirstOrDefault(it =>
it.AttributeType == typeof(TableAttribute));
                    return table == null ? it.Name :
table.ConstructorArguments[0].Value.ToString();
                }).ToList();

                selectFromInsertSql = mvData.SqlQuery;
                selectFromUpdateSql = mvData.SqlQuery;

                foreach (var name in involvedTablesNames)
                    selectFromInsertSql = selectFromInsertSql.Replace(name,
$"ct_i_{name}");

                foreach (var name in involvedTablesNames)
                    selectFromUpdateSql = selectFromUpdateSql.Replace(name,
$"ct_u_{name}");

                if (whereCondition != null)
                {
                    selectFromInsertSql =
!selectFromInsertSql.ToLower().Contains("where", StringComparison.CurrentCulture)
                        ? $"{selectFromInsertSql} where {whereCondition} "
                        : $"{selectFromInsertSql} and {whereCondition} ";

                    selectFromUpdateSql =
!selectFromUpdateSql.ToLower().Contains("where", StringComparison.CurrentCulture)
                        ? $"{selectFromUpdateSql} where {whereCondition} "
                        : $"{selectFromUpdateSql} and {whereCondition} ";
                }

                existsSubQueries = string.Empty;
```

```
                var keyMapAttrs = type.GetProperties()
                    .Where(it => it.GetCustomAttributesData().FirstOrDefault(attr
=> attr.AttributeType == typeof(KeyMapAttribute)) != null)
                    .Select(it => it.GetCustomAttributesData().FirstOrDefault(attr
=> attr.AttributeType == typeof(KeyMapAttribute)).ConstructorArguments);

                for (int i = 0; i < mvData.InvolvedTables.Count; i++)
                {
                    var table = mvData.InvolvedTables[i];
                    var props = table.GetProperties();
                    var keyProp = props.FirstOrDefault(it =>
it.GetCustomAttribute<KeyAttribute>() != null);
                    var keyPropName = keyProp == null ?
props.FirstOrDefault()?.Name : keyProp.Name;
                    var tname = involvedTablesNames[i];

                    var mappedMvColumn = keyMapAttrs.Where(it =>
it[0].Value.ToString() == tname).FirstOrDefault();

                    string mvKeyProp = mappedMvColumn == null ? keyPropName :
mappedMvColumn[1].Value.ToString();

                    existsSubQueries += @$"
                        AND NOT EXISTS (SELECT {tname}.{keyPropName} WHERE
{tname}.{keyPropName} = {mvData.Name}.{mvKeyProp}) ";
                }

                sql = $@"SELECT * ({selectFromInsertSql}
                          UNION {selectFromUpdateSql}
                          UNION SELECT * FROM {mvData.Name}
                                        WHERE {whereCondition} {existsSubQueries}
OFFSET {offset} LIMIT {limit}) AS SUBQUERY)
                          {extraOperations}
                          OFFSET {offset} LIMIT {limit}";
            }

            MVConnector.Open();

            var watch = Stopwatch.StartNew();

            var set = new DataSet();
            var command = new SqlCommand(sql, MVConnector.Connection);
            var adapter = new SqlDataAdapter(command);
            adapter.Fill(set);

            watch.Stop();
            var elapsedMinutes = watch.ElapsedMilliseconds / 1000.0 / 60.0;

            if (mvData.RefreshStrategy == RefreshStrategy.Auto && elapsedMinutes >
MVConnector.MaxAllowedSelectTime)
            {
                var syncTask = new Task(() => SyncMV(mvData.Name,
involvedTablesNames));
                syncTask.Start();
            }

            MVConnector.Close();

            return set;
        }
```

Інші методи та класи бібліотеки подані у додатку А. Модуль Internal містить класи та функціонал для роботи усередині бібліотеки, це в основному сервісні функції, що виконують лише допоміжну роботу та не мають відкритого доступу. Бібліотека може підключатись як для роботи з використанням фреймворку ASP.NET MVC, ASP.NET WebApi, так і для інших додатків, у тому числі настільних, таких, які побудовані на WPF, UWP чи MAUI, а також кросплатформених додатків, створених із використанням фреймворку Xamarin.

4.2 Результати тестування та їх аналіз

Тестування програмного засобу складається із двох основних частин: тестування продуктивності самого алгоритму вибірки удосконаленого методу матеріалізованих представлень та виконання тестів програмного коду реалізованої бібліотеки Advanced Materialization.

Для тестування продуктивності існує декілька засобів, які адекватно оцінюють показники продуктивності запиту вибірки. Одним із них є план запиту, що показує яким чином він виконувався та описує усі механізми конкретної системи керування бази даних, наприклад, поведінка оптимізатора, час виконання плану, час виконання запиту тощо.

Так як SQL є декларативною мовою, це означає, що запити описують те, що хоче користувач, а потім перетворюються на виконувані команди за допомогою оптимізатора. Ці виконувані команди відомі як плани запитів. Оптимізатор запитів створює кілька таких планів для одного запиту та визначає найефективніший.

Існують два типи планів виконання: приблизний та фактичний:

1. Фактичний план виконання означає, що він з’являється у процесі виконання запиту, під час чого відображається реальний процес виконання команд.

2. Приблизний план виконання: процесор запитів лише прогнозує точні дії, задіяні під час повернення результату. Іноді він генерується перед виконанням.

Для тестування було проведено аналіз двох планів запиту: 1 – звичайна вибірка з таблиці (рис. 4.1); 2 – вибірка удосконаленим методом із використанням аналогічного запиту (рис 4.2). Вибірка буде відбуватись 1000 публікацій від різних підписників користувача.

```
QUERY PLAN
Limit  (cost=54020271686.94..54020271709.44 rows=1000 width=148) (actual time=7684.043..7685.404 rows=1000 loops=1)
 -> Unique  (cost=54020271686.94..54021171695.94 rows=40000400 width=148) (actual time=7684.042..7685.305 rows=1000 loops=1)
    -> Sort  (cost=54020271686.94..54020371687.94 rows=40000400 width=148) (actual time=7684.041..7684.615 rows=1000 loops=1)
       Sort Key: post.created_at DESC, "user".user_id, "user".first_name, "user".last_name, post.post_id, post.title, post.description, post.reactions_count
       Sort Method: external merge  Disk: 306640kB
       -> Group  (cost=51782200083.91..54009205246.21 rows=40000400 width=148) (actual time=4406.262..6281.490 rows=2000020 loops=1)
          Group Key: post.created_at, "user".user_id, post.post_id, post.title, post.description, post.reactions_count
          -> Gather Merge  (cost=51782200083.91..54006805222.21 rows=160001600 width=148) (actual time=4406.260..5338.999 rows=2000020 loops=1)
             Workers Planned: 4
             Workers Launched: 4
             -> Group  (cost=51782199083.85..53987746489.10 rows=40000400 width=148) (actual time=4267.640..4546.938 rows=400004 loops=5)
                Group Key: post.created_at, "user".user_id, post.post_id, post.title, post.description, post.reactions_count
                -> Sort  (cost=51782199083.85..52097277284.60 rows=126031280300 width=148) (actual time=4267.637..4366.075 rows=400004 loops=5)
                   Sort Key: post.created_at DESC, "user".user_id, post.post_id, post.title, post.description, post.reactions_count
                   Sort Method: external merge  Disk: 64664kB
                   Worker 0:  Sort Method: external merge  Disk: 60168kB
                   Worker 1:  Sort Method: external merge  Disk: 61936kB
                   Worker 2:  Sort Method: external merge  Disk: 60128kB
                   Worker 3:  Sort Method: external merge  Disk: 61080kB
                   -> Nested Loop  (cost=0.84..114302994.48 rows=126031280300 width=148) (actual time=0.137..3423.900 rows=400004 loops=5)
                      Join Filter: (user_relation.following_id = "user".user_id)
                      -> Nested Loop  (cost=0.42..329653.32 rows=252057520 width=133) (actual time=0.102..2037.588 rows=400004 loops=5)
                         -> Parallel Seq Scan on post  (cost=0.00..92932.05 rows=500005 width=117) (actual time=0.017..318.701 rows=400004 loops=5)
                         -> Index Only Scan using user_relation_pkey on user_relation  (cost=0.42..0.46 rows=1 width=16) (actual time=0.004..0.004 rows=1 loops=2000020)
                            Index Cond: ((follower_id = '419c2a96-930c-456f-b624-97c27971377e'::uuid) AND (following_id = post.user_id))
                            Heap Fetches: 2000020
                      -> Index Scan using user_pkey on "user"  (cost=0.42..0.44 rows=1 width=47) (actual time=0.003..0.003 rows=1 loops=2000020)
                         Index Cond: (user_id = post.user_id)
Planning Time: 0.339 ms
Execution Time: 7720.304 ms
```

Рисунок 4.1 – План запиту звичайної вибірки

Як видно з рисунку, час виконання запиту займає приблизно 8 секунд, що є неприпустимо у сучасних веб-додатках, проте, структура створена таким чином, що немає способів при виконання редизайну сховища оптимізувати запит. Це означає, що структура даних невластива реляційній моделі, а отже, матеріалізація даних може допомогти вирішити проблему, особливо, при використанні удосконаленого методу, який базується на кешуючих таблицях, що не міститимуть великої кількості даних. При тому, як видно, пам’яті теж використовується багато.

```
QUERY PLAN
Limit  (cost=652.01..655.51 rows=200 width=1104) (actual time=3.980..4.250 rows=1000 loops=1)
 ->  Unique  (cost=652.01..655.51 rows=200 width=1104) (actual time=3.979..4.216 rows=1000 loops=1)
       ->  Sort  (cost=652.01..652.51 rows=200 width=1104) (actual time=3.978..4.013 rows=1000 loops=1)
             Sort Key: ct_i_post.created_at DESC, ct_i_user.user_id, ct_i_post.post_id, ct_i_post.title, ct_i_post.description, ct_i_post.reactions_count
             Sort Method: quicksort  Memory: 292kB
             ->  HashAggregate  (cost=642.37..644.37 rows=200 width=1104) (actual time=3.385..3.528 rows=1010 loops=1)
                   Group Key: ct_i_post.created_at, ct_i_user.user_id, ct_i_post.post_id, ct_i_post.title, ct_i_post.description, ct_i_post.reactions_count
                   ->  HashAggregate  (cost=607.34..617.35 rows=1001 width=1104) (actual time=2.765..2.939 rows=1010 loops=1)
                         Group Key: ct_i_user.user_id, ct_i_post.post_id, ct_i_post.title, ct_i_post.description, ct_i_post.reactions_count, ct_i_post.created_at
                         ->  Append  (cost=14.58..592.32 rows=1001 width=1104) (actual time=0.036..2.306 rows=1010 loops=1)
                               ->  Nested Loop  (cost=14.58..27.15 rows=1 width=1104) (actual time=0.036..0.042 rows=10 loops=1)
                                     ->  Hash Join  (cost=14.44..25.20 rows=2 width=32) (actual time=0.027..0.029 rows=1 loops=1)
                                           Hash Cond: (ct_i_user.user_id = ct_i_user_relation.following_id)
                                           ->  Seq Scan on ct_i_user  (cost=0.00..10.60 rows=60 width=16) (actual time=0.007..0.008 rows=4 loops=1)
                                           ->  Hash  (cost=14.35..14.35 rows=7 width=16) (actual time=0.013..0.013 rows=2 loops=1)
                                                 Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                                 ->  Bitmap Heap Scan on ct_i_user_relation  (cost=4.21..14.35 rows=7 width=16) (actual time=0.008..0.009 rows=2 loops=1)
                                                       Recheck Cond: (follower_id = '419c2a96-930c-456f-b624-97c27971377e'::uuid)
                                                       Heap Blocks: exact=1
                                                       ->  Bitmap Index Scan on ct_i_user_relation_pkey  (cost=0.00..4.21 rows=7 width=0) (actual time=0.005..0.005 rows=2 loops=1)
                                                             Index Cond: (follower_id = '419c2a96-930c-456f-b624-97c27971377e'::uuid)
                                     ->  Index Scan using fki_ct_i_post_user_fk on ct_i_post  (cost=0.14..0.97 rows=1 width=1104) (actual time=0.007..0.010 rows=10 loops=1)
                                           Index Cond: (user_id = ct_i_user.user_id)
                                           Filter: (created_at <= timezone('utc'::text, now()))
                               ->  Limit  (cost=51.64..550.15 rows=1000 width=117) (actual time=0.043..2.225 rows=1000 loops=1)
                                     ->  Nested Loop Anti Join  (cost=51.64..884968.43 rows=1775110 width=117) (actual time=0.043..2.184 rows=1000 loops=1)
                                           ->  Hash Anti Join  (cost=51.50..604036.37 rows=1782946 width=117) (actual time=0.037..1.666 rows=1000 loops=1)
                                                 Hash Cond: (mv_following_users_posts.user_id = ct_u_user_relation.following_id)
                                                 ->  Hash Anti Join  (cost=11.57..549011.49 rows=1975392 width=117) (actual time=0.025..1.550 rows=1000 loops=1)
                                                       Hash Cond: (mv_following_users_posts.post_id = ct_u_post.post_id)
                                                       ->  Seq Scan on mv_following_users_posts  (cost=0.00..524059.96 rows=1975559 width=117) (actual time=0.011..1.419 rows=1000 loops=1)
                                                             Filter: ((follower_id = '419c2a96-930c-456f-b624-97c27971377e'::uuid) AND (created_at <= timezone('utc'::text, now())))
                                                             Rows Removed by Filter: 4984
                                                       ->  Hash  (cost=10.70..10.70 rows=70 width=16) (actual time=0.009..0.009 rows=2 loops=1)
                                                             Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                                             ->  Seq Scan on ct_u_post  (cost=0.00..10.70 rows=70 width=16) (actual time=0.006..0.007 rows=2 loops=1)
                                                 ->  Hash  (cost=23.30..23.30 rows=1330 width=16) (actual time=0.004..0.004 rows=0 loops=1)
                                                       Buckets: 2048  Batches: 1  Memory Usage: 16kB
                                                       ->  Seq Scan on ct_u_user_relation  (cost=0.00..23.30 rows=1330 width=16) (actual time=0.004..0.004 rows=0 loops=1)
                                           ->  Index Only Scan using ct_u_post_pkey on ct_u_user  (cost=0.14..0.16 rows=1 width=16) (actual time=0.000..0.000 rows=0 loops=1000)
                                                 Index Cond: (user_id = mv_following_users_posts.user_id)
                                                 Heap Fetches: 0
Planning Time: 0.410 ms
Execution Time: 4.529 ms
```

Рисунок 4.2 – План запиту вибірки з використанням удосконаленого методу

На рисунку 4.2 видно, що запит виконується всього приблизно 5 мілісекунд. За рахунок оптимізованого використання матеріалізованого представлення вдалось досягти високих показників продуктивності. Оскільки матеріалізовані представлення призначені для швидкого доступу до даних, згенерованих складними та часозатратними запитами, алгоритм вибірки із використанням допоміжних таблиць виявився ефективним, не зважаючи на великий обсяг операцій оптимізатора, що видно із плану.

Для більш справедливого тестування було розроблено функцію, яка виконує запит декілька разів, та порівнює результати між собою і створює метричні показники, такі як, середнє арифметичне, максимальна, мінімальна оцінки тощо.

Функція аналізує результат виконання запиту та його час і на основі показників формує аналітику. Код подано у додатку А. Запустимо тестування звичайного та удосконаленого запиту, отримаємо результати, що представлені у вигляді гістограм на рисунках 4.3 та 4.4 відповідно. Виконання запиту відбувалось десять разів.

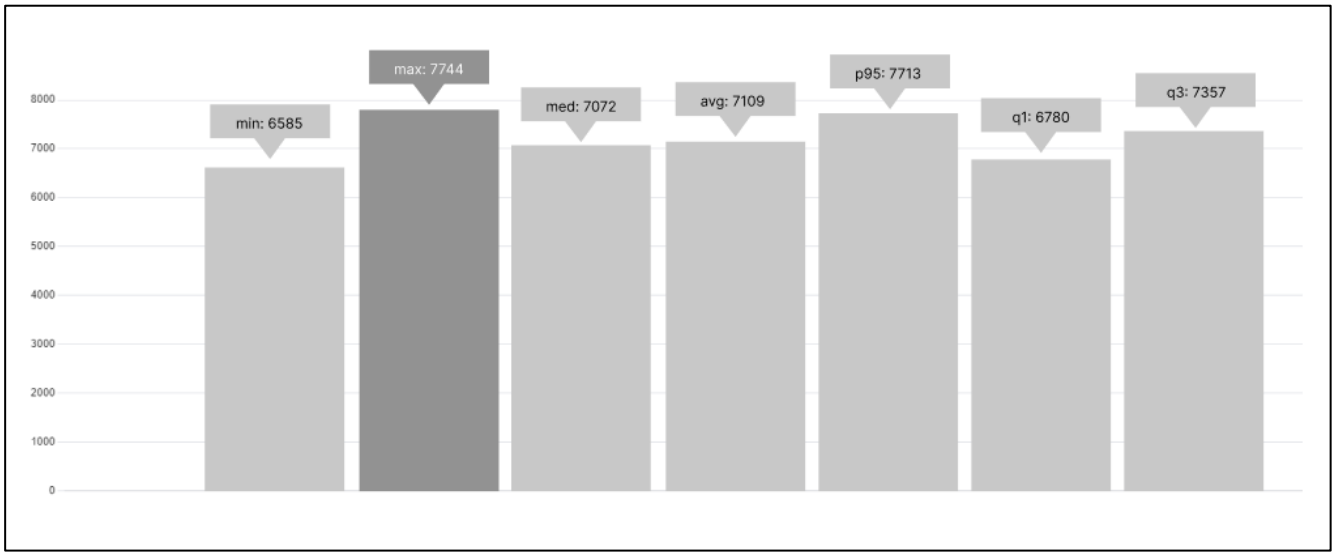


Рисунок 4.3 – Метрики виконання звичайного запиту

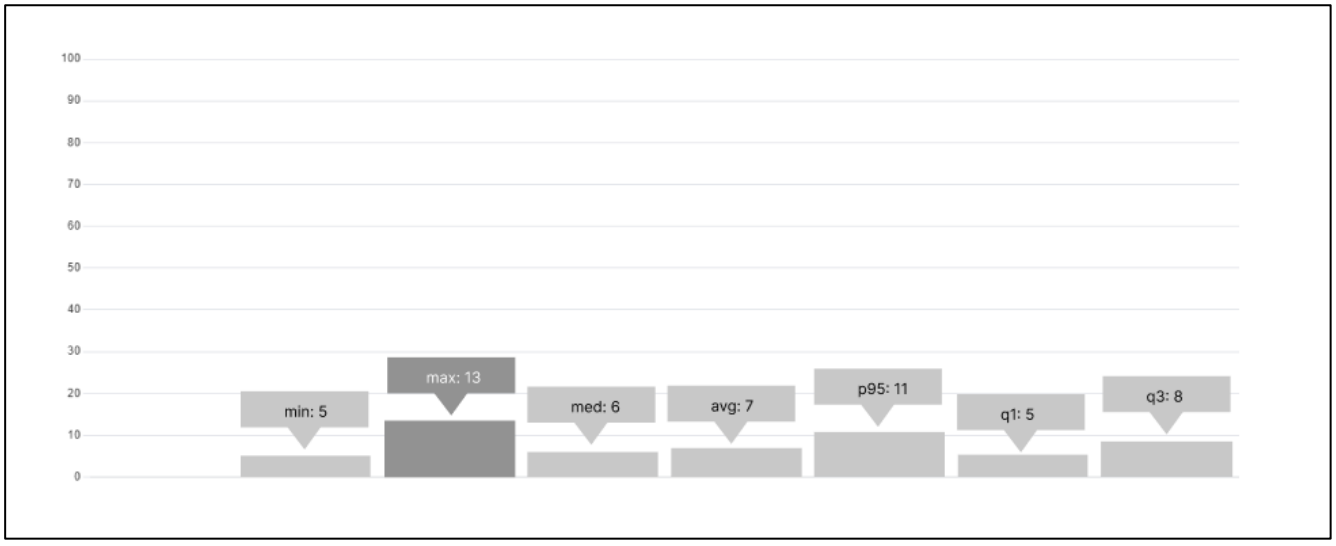


Рисунок 4.4 – Метрики виконання удосконаленого запиту

Отже, як видно, швидкість виконання запиту з використанням удосконаленого методу матеріалізованих представлень є значно швидший, ніж стандартна вибірка.

Другий етап тестування передбачає виконання модульних тексті у реалізованій бібліотеці. Для тестування було використано фреймворк XUnit, для створення об'єктів-заглушок, тобто тих, які імітують функціональність реальних об'єктів та даних, використовувався плагін Moq, а для виконання тестових порівнянь – FluentAssertions. На усі функції усіх модулів було створено по декілька юніт тестів відповідно до таблиці 4.1, де описані тест-кейси програмного засобу:

Таблиця 4.1 – Кейси модульного тестування

Ч. ч.	Модуль/Функція	Вихідні дані	Очікуваний результат
1	2	3	4
1	MVHelper_Tests Вибірка аргументів атрибуту представлення	Інформація про атрибут та тип	Має повертатись отримана інформація про атрибут: sql-запит ім'я представлення, час оновлення, стратегія оновлення та залучені таблиці
2	MVHelper_Tests Вибірка аргументів атрибуту представлення	Інформація про атрибут, що включає пустий sql запит та тип	Метод повинен викинути помилку з інформацією про помилку в аргументі атрибута
3	MVConnector_Tests Створення з'єднання з БД та ініціалізація бібліотеки	Об'єкт підключення, назва схеми даних, максимальна допустима затримка виконання запиту вибірки, прапор: чи необхідно закривати з'єднання автоматично	Метод повинен успішно встановити з'єднання та проініціалізувати бібліотеку, задавши початкові налаштування
4	MVRegistrator_Tests Створення представлення та зв'язаних допоміжних таблиць	Матеріалізоване представлення із заданими атрибутом властивостями	Метод повинен успішно створити матеріалізоване представлення та допоміжні таблиці, імена об'єктів повинні бути коректними
5	MVRegistrator_Tests Створення представлення та зв'язаних допоміжних таблиць	Матеріалізоване представлення із заданим атрибутом пустим sql-запитом	Метод повинен викинути помилку з інформацією про помилку в аргументі атрибута

Продовження таблиці 4.1

1	2	3	4
6	MVQueryExecutor_Tests Запит вибірки даних	Матеріалізоване представлення із заданими атрибутом властивостями, основна умова вибірки, додаткові операції, додаткова умова, дані для пагінації	Має повертатись DataSet, заповнений коректними даними
7	MVQueryExecutor_Tests Запит вибірки даних	Матеріалізоване представлення із некоректними значеннями у атрибуті	Метод повинен викинути помилку з інформацією про помилку в аргументі атрибута
8	MVQueryExecutor_Tests Запит вставки	Клас сутності, яку треба створити в БД	Метод повинен успішно створити сутність, відпрацювавши без помилок
9	MVQueryExecutor_Tests Запит вставки	Некоректні дані або атрибути	Метод повинен викинути помилку з інформацією про відповідну помилку
10	MVQueryExecutor_Tests Запит оновлення	Клас сутності, яку треба оновити в БД, умова, за якою необхідно оновити сутність	Метод повинен успішно оновити сутність, відпрацювавши без помилок
11	MVQueryExecutor_Tests Запит оновлення	Некоректні дані або атрибути	Метод повинен викинути помилку з інформацією про відповідну помилку
12	MVQueryExecutor_Tests Метод синхронізації	Матеріалізоване представлення	Метод повинен успішно синхронізувати дані з вихідними таблицями та оновити представлення
13	MVQueryExecutor_Tests Метод синхронізації	Матеріалізоване представлення із некоректними значеннями у атрибуті	Метод повинен викинути помилку з інформацією про відповідну помилку

У результаті тестування відповідно до кейсів, було успішно виконано та пройдено 13 тестів (рис. 4.5).

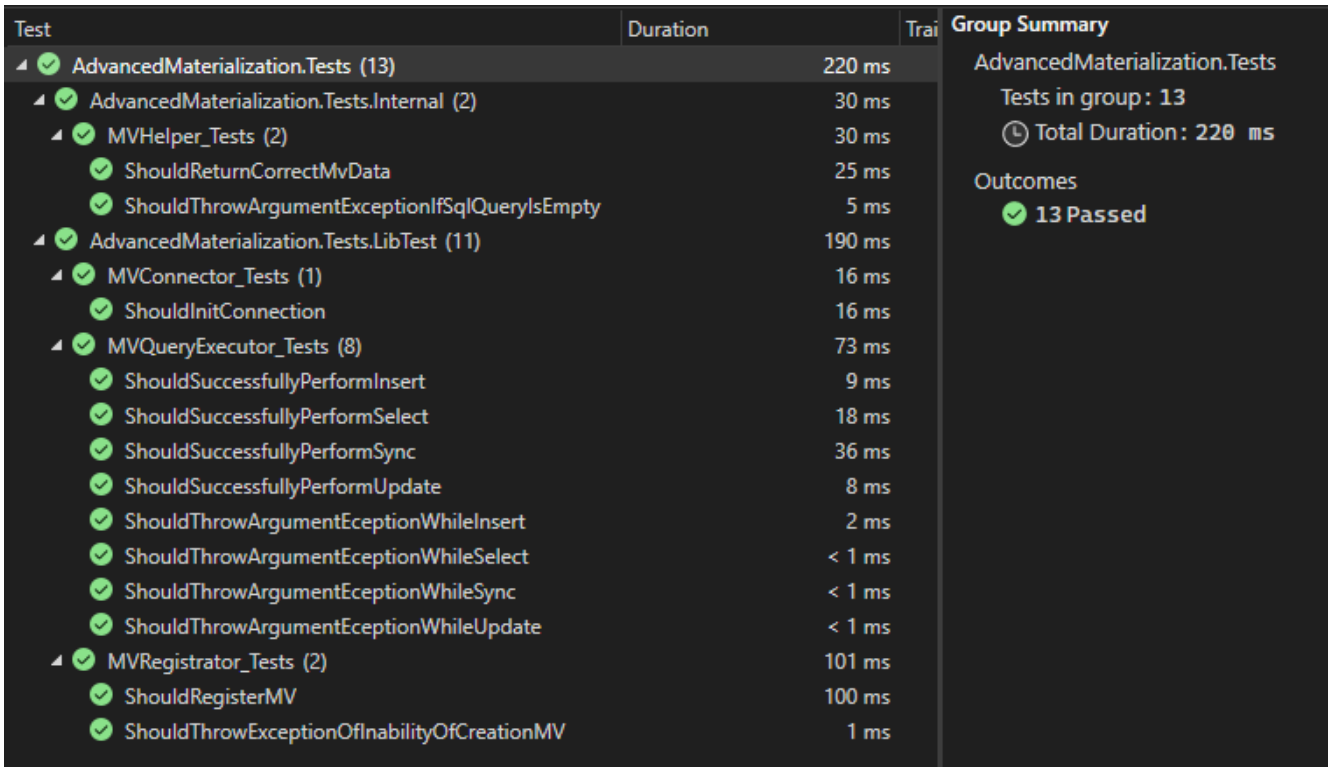


Рисунок 4.5 – Результати модульного тестування бібліотеки

4.3 Висновки

У розділі описані модулі і компоненти розроблюваної бібліотеки та їх взаємодія. Також спроектовано архітектуру програмного засобу, створено діаграми класів, що розподілені по модулям. Також розроблені діаграми діяльності основних методів та алгоритмів бібліотеки. Розглянуто реалізацію методів вставки у вихідні та допоміжні таблиці, алгоритму вибірки із використанням удосконаленого методу, а також, синхронізація змін допоміжних та вихідних таблиць і оновлення матеріалізованого представлення.

Проведено навантажувальне тестування бази даних та порівняння результатів виконання запитів вибірки із використанням удосконаленого методу матеріалізованих представлень та без. На основі них зроблено висновки про дієвість рішення та з’ясовано, що швидкодія запитів набагато збільшується у порівнянні зі звичайною вибіркою. Також створено тест-кейси, відповідно до яких проведено модульне тестування бібліотеки.

ВИСНОВКИ

У процесі дипломного проектування досліджено галузь реінжинірингу баз даних та сучасні методи і способи реінжинірингу, визначено невирішені проблеми у галузі, на основі чого удосконалено метод матеріалізованих представлень, виконано його програмну реалізацію у вигляді бібліотеки, навантажувальне тестування сховища даних та модульне тестування реалізованого плагіну.

У першому розділі дипломної роботи досліджено та проаналізовано сферу реінжинірингу сховищ даних, останні публікації, наукові статті та джерела, існуючі методи та способи проведення реінжинірингу бази даних з метою виявлення невирішених проблем. На основі аналізу визначені методологічні підходи до вирішення задачі удосконалення реінжинірингу сховищ та виконана розгорнута постановка задач дослідження.

У другому розділі роботи проаналізовано моделі, концепції та алгоритми вирішення проблем реінжинірингу баз даних. Нерозв’язані задачі запропоновано вирішити шляхом удосконалення методу матеріалізованих представлень, що полягає у використанні допоміжних таблиць для кешування вхідних змін та подальше їх застосування при синхронізації. За рахунок можливості використання матеріалізації даних у вигляді представлень на постійній основі, розроблений метод підвищує продуктивність запитів вибірки з таблиць зі складною або невластивою реляційній системі структурою організації даних.

У третьому розділі описана технологія реалізації удосконаленого методу матеріалізованих представлень. Також здійснено аналіз вимог до програмного засобу, спроектовано діаграму класів системи, а також у вигляді діаграм діяльності розроблено основні алгоритми, що застосовуються в удосконаленому методі для забезпечення функціонування основних модулів. Також обґрунтовано вибір засобу реалізації бібліотеки. Для реалізації було використано платформу .NET, мову програмування C#, плагін Stopwatch для вимірювання швидкості методів та фреймворк XUnit для проведення модульного тестування програмного засобу.

У четвертому розділі описано програмну реалізацію, навантажувальне та модульне тестування розроблюваного плагіну на основі удосконаленого методу реінжинірингу. У процесі тестування було з’ясовано, що удосконалений метод матеріалізованих представлень дає змогу суттєво підвищити показники продуктивності запитів вибірки, а також за рахунок відмови від подальшого реінжинірингу, суттєво скоротити фінансові витрати на підтримку та супровід додаткових апаратно-програмних засобів.

Шляхом реалізації нових підходів, переосмислення та комбінації існуючих алгоритмів, удосконалений метод покращує швидкість запитів вибірки даних, структура яких є невластивою для реляційної моделі, а отже, продуктивність програмної системи зростає.

Емпіричні випробування вдосконаленого підходу довели його придатність і ефективність у порівнянні з конкуруючими рішеннями у галузі реінжинірингу, а також життєздатність і функціональність програмного забезпечення, створеного з його використанням. Апробація отриманих результатів показала значний приріст продуктивності вибірки – більше, ніж на 80% у порівнянні із звичайною вибіркою.

Отже, ІТ-підприємства, які бажають підвищити продуктивність свої програмних систем, можуть використовувати реалізований програмний засіб.

Таким чином, результати дослідження за темою дипломної роботи мають наукову новизну та практичну цінність.

Результати дослідження опубліковані у фаховому науковому виданні [31].

Копії наукової публікації подані у додатку Б.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is software re-engineering process? [Electronic resource] // Shikhadeep portal. – Access mode: https://www.shikhadeep.com.np/2021/03/what-is-software-re-engineering-process.html (last access: 02.09.22). – Title from the screen.

2. Реінжиніринг програмного забезпечення [Електронний ресурс]. – Режим доступу: https://www.dreamteam.ua/ua/posluhy/reinzhynirynh-prohramnoho-zabezpechennya (дата звернення: 02.09.2022). – Назва з екрану.

3. V. Soloshchuk. Database Reengineering: Some Examples [Electronic resource] / Soloshchuk V. // LinkedIn Articles. – 2016. – Access mode: https://www.linkedin.com/pulse/database-reengineering-some-examples-vasyl-soloshchuk (last access: 05.09.22). – Title from the screen.

4. What is a Multi-Model Database? [Electronic resource] // Phoenix Global IT Services. – Access mode: https://phoenixnap.com/kb/multi-model-database (last access: 06.09.2022). – Title from the screen.

5. Y. Cao. On Optimizing Relational Self-Joins / Cao Y., Zhou Y., Chan C., Tan K. // EDBT '12: Proceedings of the 15th International Conference on Extending Database Technology. – March 2012. – P. 120-131.

6. Instagration Pt. 2: Scaling our infrastructure to multiple data centers [Electronic resource] // Instagram Engineering. – Access mode: https://instagram-engineering.com/instagration-pt-2-scaling-our-infrastructure-to-multiple-data-centers-5745cbad7834 (last access: 13.09.2022). – Title from the screen.

7. J. Jackson. Instagram Supercharges Cassandra with a Pluggable RocksDB Storage Engine [Electronic resource] / Jackson J. // Data Science, Storage. – Access mode: https://thenewstack.io/instagram-supercharges-cassandra-pluggable-rocksdb-storage-engine (last access: 14.09.2022). – Title from the screen.

8. How Instagram Uses Cassandra to Operate on a Global [Electronic resource] // DataStax. – Available from: https://www.datastax.com/blog/accelerate-rewind-how-instagram-uses-cassandra-operate-global-scale-2 (last access: 16.09.2022). – Title from

the screen.

9. Database Physical Storage: TOAST [Electronic resource] // Documentation: PostgreSQL. – Available from: https://www.postgresql.org/docs/current/storage-toast.html (last access: 20.09.2022). – Title from the screen.

10. H. Zhang. Unified SQL Query Middleware for Heterogeneous Databases / Zhang H., Zhang C., Hu R., L. Xi, D. Dongbo // Journal of Physics: Conference Series. – 2021. – P. 1-6.

11. Logstash [Electronic resource] // Logstash. – Available from: https://www.elastic.co/logstash (last access: 22.09.2022). – Title from the screen.

12. I. Holubová. Multi-model Databases: A New Journey to Handle the Variety of Data / Holubová I., Lu J. // ACM Computing Surveys. – 2019. – №3. – P. 1-38.

13. V. Kumar, K. Devi. An Architectural Framework for Constructing Materialized Views in a Data Warehouse / Kumar V., Devi K. // International Journal of Innovation, Management and Technology. – Deli, India 2013. – P. 192-196.

14. R. Chirkova. Materialized Views / Chirkova R., Yang J. // Foundations and Trends in Databases. – Raleigh, North Carolina, USA, 2012. – P.10-16.

15. pgbench [Electronic resource] // Documentation: PostgreSQL. – Available from: https://www.postgresql.org/docs/current/pgbench.html (last access: 28.09.2022). – Title from the screen.

16. Apache JMeter [Electronic resource] // Apache software foundation. – Available from: https://jmeter.apache.org/index.html (last access: 28.09.2022). – Title from the screen.

17. Y. Unal. Migration of Data from Relational Database to Graph Database / Unal Y., Oguztuzun H. // International Conference on Information and Software Technologies. – Istanbul, Turkey, 2018. – P. 1-5.

18. M. Khan. Exploring Query Optimization Techniques in Relational Databases / Khan M., Khan M. N. A. // International Journal of Database Theory and Application. – China, 2013. – P. 11-20.

19. J. Habimana. Query Optimization Techniques - Tips For Writing Efficient And Faster SQL Queries / Habimana J. // International Journal Of Scientific & Technology

Research. – India, 2015. – № 10. – P. 22-25.

20. S. Mukherjee. Indexes in Microsoft SQL Server / Mukherjee S. // Cornel University. – Chicago, USA, 2019. – P. 1-16.

21. A. Abdulghafor. An Integrated Indexing Approach to Improve Query Processing / Abdulghafor A., Kamsuriah A. // International Conference on Electrical Engineering and Informatics (ICEEI). – Kuala Terengganu, Malaysia, November 2021.

22. G. L. Sanders. Denormalization Effects on Performance of RDBMS / Sanders G. L. // Proceedings of the 34th Annual Hawaii International Conference. – USA, 2001.

23. H. Duan. Incremental Materialized View Maintenance on Distributed Log Structured Merge-Tree / Duan H., Hu H., Qian W., Ma H., Wang X., Zhou A. // Database Systems for Advanced Applications. – May 2018. – P. 682–700.

24. J. Zhou. Lazy Maintenance of Materialized Views / Zhou J., Larson P.-A., Elmongui H. G. // Proceedings of the 33rd international conference on Very large data bases. – USA, 2007. – P. 231-242.

25. Postgresql Count(*) Made Fast [Electronic resource] // CYBERTEC. – Available from: https://www.cybertec-postgresql.com/en/postgresql-count-made-fast (last access: 02.10.2022). – Title from the screen.

26. R. Virgilio. Converting Relational to Graph Databases / Virgilio R., Maccioni A., Torlone R. // First International Workshop on Graph Data Management Experiences and Systems. – Rome, Italy, 2013. – P. 1-6.

27. Multimodel database [Electronic resource]. – Available from: https://www.techtarget.com/searchdatamanagement/definition/multimodel-database (last access: 04.10.2022). – Title from the screen.

28. M.-C. Albutiu. Massively Parallel Sort-Merge Joins in Main Memory Multi Core Database Systems / Albutiu M.-C., Kemper A., Neuman T. // Proceedings of the VLDB Endowment. – Germany, 2012. – №10. – P. 1064-1075.

29. D. Comer. Computing Surveys: The Ubiquitous B-Tree / Comer D. // ACM Computing Surveys. – West Lafayette, Indiana, USA, June 1979. – №2 – P. 127-137.

30. T-SQL commands performance comparison – NOT IN vs SQL NOT EXISTS vs SQL LEFT JOIN vs SQL EXCEPT [Electronic resource] // SQLShack. – Available

form: https://www.sqlshack.com/t-sql-commands-performance-comparison-not-vs-not-exists-vs-left-join-vs-except (last access: 10.10.2022). – Title from the screen.

31. Бойко В. О., Форкун Ю. В. Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних // Вимірювальна та обчислювальна техніка в технологічних процесах. – Хмельницький, 2022. – №3 (71). – С. 87-91.

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А.1 Програмний код класу MVConnector

```
using Microsoft.Data.SqlClient;
using System.Data;

namespace AdvancedMaterialization.Lib
{
    /// <summary>
    /// Connector to use Advanced Materialization Library
    /// </summary>
    public static class MVConnector
    {
        /// <summary>
        /// Database Scheme
        /// </summary>
        public static string Scheme { get; private set; }

        /// <summary>
        /// Sql Connection
        /// </summary>
        public static SqlConnection Connection { get; private set; }

        /// <summary>
        /// Show what the maximum time should be for select responses. It will be
used to determine how often refresh materialized views
        /// </summary>
        public static double? MaxAllowedSelectTime { get; private set; }

        /// <summary>
        /// Close connection automatically by library or manually
        /// </summary>
        public static bool? AutoCloseConnection { get; private set; }

        /// <summary>
        /// Init Connection to use Advanced Materialization Library
        /// </summary>
        /// <param name="connection"></param>
        /// <param name="scheme"></param>
        /// <param name="maxAllowedSelectTime"></param>
        public static void Init(SqlConnection connection, string scheme =
"public", double maxAllowedSelectTime = 1.0, bool autoCloseConnection = false)
        {
            Connection ??= connection;
            Scheme ??= scheme;
            MaxAllowedSelectTime ??= maxAllowedSelectTime;
            AutoCloseConnection ??= autoCloseConnection;
        }

        internal static void Open()
        {
```

```
            if (Connection.State != ConnectionState.Open)
                Connection.Open();
        }

        internal static void Close()
        {
            if (AutoCloseConnection == true)
                Connection.Close();
        }
    }
}
```

A.2 Програмний код класу MVRegistrator

```
using AdvancedMaterialization.Lib.Attributes;
using AdvancedMaterialization.Lib.Internal;
using Microsoft.Data.SqlClient;
using System.ComponentModel.DataAnnotations.Schema;

namespace AdvancedMaterialization.Lib
{
    /// <summary>
    /// Registrator for creation specified materialized views with caching tables
    /// </summary>
    public static class MVRegistrator
    {
        /// <summary>
        /// Creates specified materialized views with caching tables if they not
exist
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <exception cref="ArgumentException"></exception>
        public static void Register<T>() where T : class
        {
            var type = typeof(T);

            var mvAttribute = MVHelper.GetMaterializedViewAttribute(type);

            var attributeData = MVHelper.ParseAttribute(mvAttribute, typeof(T));
            var connection = MVConnector.Connection;

            MVConnector.Open();

            string createMVSql =
                @$"CREATE MATERIALIZED VIEW IF NOT EXISTS
{MVConnector.Scheme}.{attributeData.Name} AS {attributeData.SqlQuery} WITH DATA";

            var command = new SqlCommand(createMVSql, connection);
            command.ExecuteNonQuery();

            MVHelper.CreateIndexes(type.GetProperties().ToList());

            if (attributeData.InvolvedTables.Count == 0)
            {
                return;
            }

            foreach (var t in attributeData.InvolvedTables)
            {
                var table = t.GetCustomAttributesData().FirstOrDefault(it =>
it.AttributeType == typeof(TableAttribute));
```

```
                var tableName = t != null ?
table.ConstructorArguments[0].Value.ToString() : t.Name;

                var foreignKeys = new List<string>();

                var props = t.GetProperties().Select(it =>
                {
                    var attrData = it.GetCustomAttributesData().FirstOrDefault(it
=> it.AttributeType == typeof(ColumnAttribute));
                    var attrForeignKey =
it.GetCustomAttributesData().FirstOrDefault(it => it.AttributeType ==
typeof(ForeignKeyAttribute));

                    if (attrForeignKey != null &&
attributeData.InvolvedTables.Contains(it.PropertyType))

foreignKeys.Add(attrForeignKey.ConstructorArguments[0].Value.ToString());


                    return it;
                }).ToList();

                MVHelper.CreateTable($"ct_i_{tableName}", props, foreignKeys);
                MVHelper.CreateTable($"ct_u_{tableName}", props);
            }

            MVConnector.Close();
        }
    }
}
```

A.3 Програмний код класу MVQueryRegistator

```
using AdvancedMaterialization.Lib.Attributes;
using AdvancedMaterialization.Lib.Internal;
using Microsoft.Data.SqlClient;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data;
using System.Diagnostics;
using System.Reflection;

namespace AdvancedMaterialization.Lib
{
    public static class MVQueryExecutor
    {
        /// <summary>
        /// Returns the DataSet that represents the retrieved data from
materialized view and caching table
        /// </summary>
        /// <typeparam name="TView"></typeparam>
        /// <param name="whereCondition"></param>
        /// <param name="offset"></param>
        /// <param name="limit"></param>
        /// <param name="extraOperations"></param>
        /// <returns></returns>
        public static DataSet MVSelect<TView>(string whereCondition = null, int
offset = 0, int limit = 1000, string extraOperations = null) where TView : class
        {
            var type = typeof(TView);
```

```
            var mvAttribute = MVHelper.GetMaterializedViewAttribute(type);
            var mvData = MVHelper.ParseAttribute(mvAttribute, type);

            string selectFromInsertSql = null;
            string selectFromUpdateSql = null;
            string existsSubQueries = null;
            string sql = $"SELECT * FROM {mvData.Name} WHERE {whereCondition}
{extraOperations} OFFSET {offset} LIMIT {limit}";
            var involvedTablesNames = new List<string>();

            if (mvData.InvolvedTables.Count != 0)
            {
                involvedTablesNames = mvData.InvolvedTables.Select(it =>
                {
                    var table = it.GetCustomAttributesData().FirstOrDefault(it =>
it.AttributeType == typeof(TableAttribute));
                    return table == null ? it.Name :
table.ConstructorArguments[0].Value.ToString();
                }).ToList();

                selectFromInsertSql = mvData.SqlQuery;
                selectFromUpdateSql = mvData.SqlQuery;

                foreach (var name in involvedTablesNames)
                    selectFromInsertSql = selectFromInsertSql.Replace(name,
$"ct_i_{name}");

                foreach (var name in involvedTablesNames)
                    selectFromUpdateSql = selectFromUpdateSql.Replace(name,
$"ct_u_{name}");

                if (whereCondition != null)
                {
                    selectFromInsertSql =
!selectFromInsertSql.ToLower().Contains("where", StringComparison.CurrentCulture)
                        ? $"{selectFromInsertSql} where {whereCondition} "
                        : $"{selectFromInsertSql} and {whereCondition} ";

                    selectFromUpdateSql =
!selectFromUpdateSql.ToLower().Contains("where", StringComparison.CurrentCulture)
                        ? $"{selectFromUpdateSql} where {whereCondition} "
                        : $"{selectFromUpdateSql} and {whereCondition} ";
                }

                existsSubQueries = string.Empty;

                var keyMapAttrs = type.GetProperties()
                    .Where(it => it.GetCustomAttributesData().FirstOrDefault(attr
=> attr.AttributeType == typeof(KeyMapAttribute)) != null)
                    .Select(it => it.GetCustomAttributesData().FirstOrDefault(attr
=> attr.AttributeType == typeof(KeyMapAttribute)).ConstructorArguments);

                for (int i = 0; i < mvData.InvolvedTables.Count; i++)
                {
                    var table = mvData.InvolvedTables[i];
                    var props = table.GetProperties();
                    var keyProp = props.FirstOrDefault(it =>
it.GetCustomAttribute<KeyAttribute>() != null);
                    var keyPropName = keyProp == null ?
props.FirstOrDefault()?.Name : keyProp.Name;
                    var tname = involvedTablesNames[i];
```

```
                    var mappedMvColumn = keyMapAttrs.Where(it =>
it[0].Value.ToString() == tname).FirstOrDefault();

                    string mvKeyProp = mappedMvColumn == null ? keyPropName :
mappedMvColumn[1].Value.ToString();

                    existsSubQueries += @$"
                        AND NOT EXISTS (SELECT {tname}.{keyPropName} WHERE
{tname}.{keyPropName} = {mvData.Name}.{mvKeyProp}) ";
                }

                sql = $@"SELECT * ({selectFromInsertSql}
                          UNION {selectFromUpdateSql}
                          UNION SELECT * FROM {mvData.Name}
                                        WHERE {whereCondition} {existsSubQueries}
OFFSET {offset} LIMIT {limit}) AS SUBQUERY)
                          {extraOperations}
                          OFFSET {offset} LIMIT {limit}";
            }

            MVConnector.Open();

            var watch = Stopwatch.StartNew();

            var set = new DataSet();
            var command = new SqlCommand(sql, MVConnector.Connection);
            var adapter = new SqlDataAdapter(command);
            adapter.Fill(set);

            watch.Stop();
            var elapsedMinutes = watch.ElapsedMilliseconds / 1000.0 / 60.0;

            if (mvData.RefreshStrategy == RefreshStrategy.Auto && elapsedMinutes >
MVConnector.MaxAllowedSelectTime)
            {
                var syncTask = new Task(() => SyncMV(mvData.Name,
involvedTablesNames));
                syncTask.Start();
            }

            MVConnector.Close();

            return set;
        }

        /// <summary>
        /// Inserts data into table and additional cahching tables
        /// </summary>
        /// <typeparam name="TTable"></typeparam>
        /// <param name="entity"></param>
        public static void Insert<TTable>(TTable entity) where TTable : class
        {
            var data = MVHelper.GetEntityData(entity);
            var columnsStr = $"({string.Join(',', data.Columns)})";
            var valuesStr = $"({string.Join(',', data.Values)})";

            MVConnector.Open();

            var sqlTran = MVConnector.Connection.BeginTransaction();
            var command = MVConnector.Connection.CreateCommand();
            command.Transaction = sqlTran;
```

```
            command.CommandText = $"INSERT INTO
{MVConnector.Scheme}.{data.TableName} {columnsStr} VALUES {valuesStr}; ";
            command.ExecuteNonQuery();

            if (data.RelatedMV != null)
            {
                command.CommandText = $"INSERT INTO
{MVConnector.Scheme}.ct_i_{data.TableName} {columnsStr} VALUES {valuesStr}; ";
                command.ExecuteNonQuery();

                command.CommandText = $"INSERT INTO
{MVConnector.Scheme}.ct_u_{data.TableName} {columnsStr} VALUES {valuesStr}; ";
                command.ExecuteNonQuery();
            }

            sqlTran.Commit();

            MVConnector.Close();
        }

        /// <summary>
        /// Updates data in table and additional cahching tables
        /// </summary>
        /// <typeparam name="TTable"></typeparam>
        /// <param name="entity"></param>
        /// <param name="condtion"></param>
        public static void Update<TTable>(TTable entity, string condtion) where
TTable : class
        {
            var data = MVHelper.GetEntityData(entity);
            var sets = data.Columns.Select((col, i) => $"{col} =
{data.Values[i]}");

            MVConnector.Open();

            var sqlTran = MVConnector.Connection.BeginTransaction();
            var command = MVConnector.Connection.CreateCommand();
            command.Transaction = sqlTran;

            command.CommandText = $"UPDATE {MVConnector.Scheme}.{data.TableName}
SET {string.Join(',', sets)} WHERE {condtion}; ";
            command.ExecuteNonQuery();

            if (data.RelatedMV != null)
            {
                string ctICondition = condtion.Replace(data.TableName,
$"ct_i_{data.TableName}");
                command.CommandText = $"UPDATE
{MVConnector.Scheme}.ct_i_{data.TableName} SET {string.Join(',', sets)} WHERE
{ctICondition}; ";
                command.ExecuteNonQuery();

                string ctUCondition = condtion.Replace(data.TableName,
$"ct_u_{data.TableName}");
                command.CommandText = $"UPDATE
{MVConnector.Scheme}.ct_u_{data.TableName} SET {string.Join(',', sets)} WHERE
{ctUCondition}; ";
                command.ExecuteNonQuery();
            }

            sqlTran.Commit();

            MVConnector.Close();
```

```
        }

        /// <summary>
        /// Refreshes materialized view and synchronizes caching tables
        /// </summary>
        /// <typeparam name="T"></typeparam>
        public static void Sync<TView>() where TView : class
        {
            var type = typeof(TView);

            var mvAttribute = MVHelper.GetMaterializedViewAttribute(type);
            var mvData = MVHelper.ParseAttribute(mvAttribute, type);

            var tableNames = mvData.InvolvedTables.Select(it =>
            {
                var table = it.GetCustomAttributesData().FirstOrDefault(it =>
it.AttributeType == typeof(TableAttribute));
                return table == null ? it.Name :
table.ConstructorArguments[0].Value.ToString();
            }).ToList();

            SyncMV(mvData.Name, tableNames);
        }

        private static void SyncMV(string mvName, List<string> tableNames)
        {
            MVConnector.Open();

            var sqlTran = MVConnector.Connection.BeginTransaction();
            var command = MVConnector.Connection.CreateCommand();
            command.Transaction = sqlTran;

            command.CommandText = $"REFRESH MATERIALIZED VIEW CONCURRENTLY
{mvName}";
            command.ExecuteNonQuery();

            foreach (string table in tableNames)
            {
                command.CommandText = $"DELETE FROM ct_i_{table}";
                command.ExecuteNonQuery();

                command.CommandText = $"DELETE FROM ct_u_{table}";
                command.ExecuteNonQuery();
            }

            sqlTran.Commit();

            MVConnector.Close();
        }
    }
}
```

А.4 Програмний код класу MVHelper

```
using AdvancedMaterialization.Lib.Attributes;
using System.Collections.ObjectModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Reflection;

namespace AdvancedMaterialization.Lib.Internal
```

```
{
    public static class MVHelper
    {
        public static MVData ParseAttribute(CustomAttributeData mvAttribute, Type
type)
        {
            string mvQuery = mvAttribute.ConstructorArguments[0].Value.ToString();

            if (string.IsNullOrEmpty(mvQuery))
                throw new ArgumentException($"Cannot create or access the
materialized view {type.Name} because the provided SqlQuery is null or empty.");

            string mvName = string.Empty;
            TimeSpan mvRefeshTime = TimeSpan.Zero;
            RefreshStrategy mvRefreshStrategy = RefreshStrategy.Auto;
            List<Type> involvedTables = new();

            foreach (var arg in mvAttribute.NamedArguments)
            {
                var value = arg.TypedValue.Value;

                switch (arg.MemberName)
                {
                    case "Name":
                        string mvNameValue = value?.ToString();
                        mvName = string.IsNullOrEmpty(mvNameValue) ? type.Name :
mvNameValue;
                        break;
                    case "RefreshTime":
                        mvRefeshTime = (TimeSpan)value;
                        break;
                    case "RefreshStrategy":
                        mvRefreshStrategy = (RefreshStrategy)value;
                        break;
                    case "InvolvedTables":
                        var data =
(ReadOnlyCollection<CustomAttributeTypedArgument>)value;
                        involvedTables = data.Select(it =>
(Type)it.Value).ToList();
                        break;
                }
            }

            return new MVData
            {
                SqlQuery = mvQuery,
                Name = mvName,
                RefreshTime = mvRefeshTime,
                RefreshStrategy = mvRefreshStrategy,
                InvolvedTables = involvedTables
            };
        }

        internal static CustomAttributeData GetMaterializedViewAttribute(Type t)
        {
            if (MVConnector.Connection == null)
                throw new ArgumentException($"Cannot create or access the
materialized view {t.Name} because the Advanced Materialization Library is not
initialized.");

            var customAttributesData = t.GetCustomAttributesData();

            if (customAttributesData == null || customAttributesData.Count == 0)
```

```
                throw new ArgumentException($"Cannot create or access the
materialized view {t.Name} because the MaterializedBiewAttribute on this type does
not exist.");

            var mvAttribute = customAttributesData.Where(it => it.AttributeType ==
typeof(MaterializedViewAttribute)).FirstOrDefault();

            if (mvAttribute == null)
                throw new ArgumentException($"Cannot create or access the
materialized view {t.Name} because the MaterializedBiewAttribute on this type does
not exist.");

            return mvAttribute;
        }

        internal static MVTableData GetEntityData<TTable>(TTable entity) where
TTable : class
        {
            var type = typeof(TTable);
            var attributes = type.GetCustomAttributesData();
            var mvMappedAttr = attributes.FirstOrDefault(it => it.AttributeType ==
typeof(MaterializedViewMapAttribute));

            var mvAttribute =
GetMaterializedViewAttribute(mvMappedAttr.ConstructorArguments[0].Value as Type);
            var mvData = ParseAttribute(mvAttribute, type);

            var tableAttr = attributes.FirstOrDefault(it => it.AttributeType ==
typeof(TableAttribute));
            var tableName = tableAttr == null ? type.Name :
tableAttr.ConstructorArguments[0].Value.ToString();

            var columns = new List<string>();
            var values = new List<string>();


            foreach (var column in type.GetProperties())
            {
                var columnAttr =
column.GetCustomAttributesData().FirstOrDefault(it =>
                    it.AttributeType == typeof(ColumnAttribute) ||
                    it.AttributeType == typeof(ForeignKeyAttribute));

                var columnName = columnAttr == null ? column.Name :
columnAttr.ConstructorArguments[0].Value.ToString();
                var columnValue = column.GetValue(entity);
                var columnValueType = columnValue.GetType();

                if (columnValueType.IsClass)
                {
                    var innerProps = columnValueType.GetProperties();
                    var innerKeyProp = innerProps.FirstOrDefault(it =>
                        it.GetCustomAttributesData().FirstOrDefault(attr =>
attr.AttributeType == typeof(KeyAttribute)) != null);

                    if (innerKeyProp != null)
                    {
                        columnValue = innerKeyProp.GetValue(columnValue);
                    }
                }

                var columnValueStr = columnValueType ==
```

```
                    typeof(string) || columnValueType == typeof(char) ||
columnValueType == typeof(TimeSpan) ||
                    columnValueType == typeof(Guid) || columnValueType ==
typeof(DateTime)
                    ? $"'{columnValue}'" : columnValue.ToString();

                columns.Add(columnName);
                values.Add(columnValueStr);
            }

            return new MVTableData
            {
                TableName = tableName,
                Columns = columns,
                Values = values,
                RelatedMV = mvData,
            };
        }
    }
}
```

A.5 Програмний код класу MVData

```
namespace AdvancedMaterialization.Lib.Internal
{
    public class MVData
    {
        public string SqlQuery { get; init; }
        public string Name { get; init; }
        public TimeSpan RefreshTime { get; init; }
        public RefreshStrategy RefreshStrategy { get; init; }
        public List<Type> InvolvedTables { get; init; }
    }

    public class MVTableData
    {
        public MVData RelatedMV { get; init; }
        public string TableName { get; init;}
        public List<string> Values { get; init; }
        public List<string> Columns { get; init; }
    }
}
```

A.6 Програмний код класу MaterializedViewAttribute

```
namespace AdvancedMaterialization.Lib.Attributes
{
    /// <summary>
    /// Attribute to mark a class that represents the specified materialized view
    /// </summary>
    [AttributeUsage(AttributeTargets.Class)]
    public sealed class MaterializedViewAttribute : Attribute
    {
        /// <summary>
        /// Materialized View Name<br/>
        /// If name is not specified the name of a class will be used instead
        /// </summary>
        public string Name { get; init; }
```

```
        /// <summary>
        /// Time when refresh specified materialized view<br/>
        /// Default: 1 day
        /// </summary>
        public TimeSpan RefreshTime { get; init; } = new(1, 0, 0, 0);

        /// <summary>
        /// Refresh strategy<br/>
        /// Default: Auto
        /// </summary>
        public RefreshStrategy RefreshStrategy { get; init; } =
RefreshStrategy.Auto;

        /// <summary>
        /// Tables are using in sql query to fill specified materialized view<br/>
        /// Default: Empty array of table types
        /// </summary>
        public Type[] InvolvedTables { get; init; } = Array.Empty<Type>();

        /// <summary>
        /// Constructor of MaterializedViewAttribute
        /// </summary>
        /// <param name="sqlQuery">Sql query used to fill specified materialized
view</param>
        public MaterializedViewAttribute(string sqlQuery)
        {
        }
    }
}
```

А.7 Програмний код класу KeyMapAttribute

```
namespace AdvancedMaterialization.Lib.Attributes
{
    /// <summary>
    /// Uses in materialized views to specify the origin table and column name of
columns
    /// </summary>
    [AttributeUsage(AttributeTargets.Property)]
    public class KeyMapAttribute : Attribute
    {
        public KeyMapAttribute(string mappedTableName, string mappedColumnName)
        {
        }
    }
}
```

А.8 Програмний код класу IndexAttribute

```
namespace AdvancedMaterialization.Lib.Attributes
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexAttribute : Attribute
    {
        /// <summary>
        /// Extra columns to include into index, separated by comma
        /// </summary>
```

```
        public string ExtraColumns { get; set; }

        /// <summary>
        /// Constructor of IndexAttribute
        /// </summary>
        /// <param name="type">Index type<br/> Default: Btree</param>
        public IndexAttribute(IndexType type = IndexType.Btree)
        {
        }
    }
}
```

А.9 Програмний код класу MaterializedViewMapAttribute

```
namespace AdvancedMaterialization.Lib.Attributes
{
    [AttributeUsage(AttributeTargets.Class)]
    public class MaterializedViewMapAttribute : Attribute
    {
        public MaterializedViewMapAttribute(Type[] mvTypes)
        {
        }

        public MaterializedViewMapAttribute(Type mvType)
        {
        }
    }
}
```

ДОДАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

ISSN 2219-9365
DOI: 10.31891/2219-9365

Міжнародний науково-технічний журнал

ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

2022, № 3

International scientific-technical journal

MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES

2022, Issue 3

Хмельницький 2022
Khmelnytskyi 2022

МІЖНАРОДНИЙ НАУКОВО-ТЕХНІЧНИЙ ЖУРНАЛ
ВИМІРЮВАЛЬНА ТА ОБЧИСЛЮВАЛЬНА ТЕХНІКА В ТЕХНОЛОГІЧНИХ ПРОЦЕСАХ

Затверджений як фахове видання (перереєстрація), група «Б»
Наказ МОН 28.12.2019 №1643

Засновано в травні 1997 р. ***Виходить 4 рази на рік***

Хмельницький, 2022, № 3 (71)

Засновник і видавець: **Хмельницький національний університет (до 2005 р. — Технологічний університет Поділля, м. Хмельницький)**

Наукова бібліотека України ім. В.І. Вернадського http://nbuv.gov.ua/j-tit/vott

Журнал включено до наукометричних баз:

Index Copernicus http://jml2012.indexcopernicus.com/.p24781565,3.html
Google Schoolar http://scholar.google.com.ua/citations?user=nwN_nusAAAAJ&hl=uk
CrossRef http://doi.org/10.31891/2219-9365

Рекомендовано до друку рішенням Вченої ради Хмельницького національного університету, протокол № 02 від 29.09.2022

Адреса редакції: **Україна, 29016, м. Хмельницький, вул. Інститутська, 11, Хмельницький національний університет, Редакція журналу "Вимірювальна та обчислювальна техніка в технологічних процесах"**

☎ **067-347-74-57**

e-mail: **vottp@khmnu.edu.ua**

web: **http://vottp.khmnu.edu.ua**

Зареєстровано Міністерством України у справах преси та інформації.
Свідоцтво про державну реєстрацію друкованого засобу масової інформації
Серія КВ № 24923-14863 ПР від 12 липня 2021 року (перереєстрація)

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

ЗМІСТ

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

https://doi.org/10.31891/2219-9365-2022-71-3-10
УДК 004.65

ВЯЧЕСЛАВ БОЙКО
Хмельницький національний університет
e-mail: bviacheslav.12@gmail.com
ЮРІЙ ФОРКУН
Хмельницький національний університет
https://orcid.org/0000-0002-7906-4191
e-mail: forkun@ridne.net

УДОСКОНАЛЕННЯ МЕТОДУ МАТЕРІАЛІЗОВАНИХ ПРЕДСТАВЛЕНЬ У РЕІНЖИНІРИНГУ БАЗИ ДАНИХ

Розглянуто основні методи реінжинірингу бази даних у сучасних програмних системах. Виконано порівняння та аналіз найбільш популярних та окреслено основні етапи реінжинірингу сховищ. Удосконалено метод матеріалізованих представлень, який використовується на етапі редизайну табличної структури бази даних, що дозволяє застосовувати представлення як один із оптимізаційних компонентів на постійній основі, а отже, і значно підвищує продуктивність запитів вибірки даних у програмній системі.
Ключові слова: реінжиніринг, редизайн, декомпозиція, база даних, матеріалізоване представлення.

VIACHESLAV BOIKO, YURIY FORKUN
Khmelnitsky National University

IMPROVING THE METHOD OF MATERIALIZATION VIEWS AS A PART OF A DATABASE REENGINEERING PROCESS

Data warehouse – is the most important component in the modern cloud web-applications. It has a significant impact on performance of the program system. According to its influence on whole system productivity, incorrect database schemas, none-optimized queries or unsuitable data organization structure can greatly reduce the performance of whole application. With a constantly growing amount of information and brand-new customer requirements that possibly can not be applied to the current architecture, the reengineering process is a strong necessary to refactor and reform the system architecture including data storages. There are a variety of reengineering methods that could make the system faster, easier to maintain and more stable. But some of them are very cost-expensive due to transforming into a more complex formation that it was before, thus maintenance of such system can be not so easy. On the other hand, some of methods are not able to give enough results to fit customer performance requirements. Because of that, solution should use the known database features with rethinking of its usage as a part of the reengineering process. The potential mechanism of increasing the database performance can be the relational database objects – materialized views with their own refreshing strategies. Having a big read performance, materialized views can be very useful to accessing a large amount of data, but due to some refresh constraints they cannot be used permanently. So, in the paper the main methods of database reengineering in modern software systems are considered. A comparison and analysis of the most popular ones was performed, and the main stages of data storage reengineering were outlined. The method of materialized views, which is used at the stage of redesigning the tabular structure of the database, has been improved, which allows use the view as one of the optimization components on a permanent basis, thus significantly increases the performance of data selection queries in the software system.
Keywords: reengineering, redesign, decomposition, database, materialized view.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Більшість сучасних програмних систем мають проблеми зі швидкістю обробки даних через постійно зростаючий обсяг інформації. Нові бізнес-вимоги вимагають реалізацію нових функціональних можливостей, які часто розробляються із використанням застарілих шаблонів та структур. Інформаційне сховище є основним артефактом впливу на продуктивність програмного забезпечення, оскільки характер та структура організації інформації є найвагомішими чинниками, що впливають на швидкість обробки даних. Для підвищення показників продуктивності застосовують різні методи реінжинірингу сховища даних, більшість з яких хоч і вирішують проблеми продуктивності, проте підтримка та супровід таких систем є дороговартісним та складним процесом. А тому аналіз та удосконалення існуючих методів реінжинірингу бази даних є актуальним насьогодні.

Зазвичай реінжиніринг проводиться поступово у декілька етапів: редизайну, декомпозиції та міграції. Проте, якщо необхідні показники приросту продуктивності та фінансових витрат уже досягнуті, процес реінжинірингу вважається завершеним на конкретному етапі лише тоді, якщо при подальших кроках показники погіршуватимуться або залишатимуться незмінними.

Крім того, процес редизайну вважається обов'язковим етапом. При цьому не завжди дає змогу отримати бажані результати. Варто зазначити, що за основу реінжинірингу взято сховище реляційного типу, а тому усі описані методи можуть застосовуватись лише до такого типу баз даних.

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

Аналіз досліджень та публікацій

На етапі редизайну виділяють декілька часто використовуваних методів, що дозволяють підвищити продуктивність роботи з базою даних. У статті [1] описані основні методи оптимізації запитів. Крім того, представлено метод денормалізації лічильників. Перевагою є те, що вибірка вже попередньо обрахованих значень дозволяє уникнути затратних запитів для підрахунку записів. Проте недоліком є неможливість застосування його до динамічного текстового пошуку.

Перебудова індексів також є одним із основних методів редизайну бази даних. У публікації [2] запропоновано вирішення проблеми автономної роботи із фрагментованими індексами в реляційній СУБД шляхом відстеження поточного стану індексів і, якщо вони стають фрагментованими, автоматично перебудовувати їх. Перевагами методу є те, що заходи із перебудови індексів, які проводяться на постійній основі дозволяють прискорити операції у базах даних.

Матеріалізація як метод реінжинірингу може значно скоротити час обробки запитів, особливо для агрегаційних запитів до великих таблиць, оскільки матеріалізовані представлення зберігають результат вихідного запиту, що є перевагою. У статті [3] описано використання представлення для оптимізації вибірки. Проте, недоліком є те, що застосування матеріалізованих представлень на постійній основі може призвести до проблем з продуктивністю, оскільки їх оновлення займає багато часу, а також у деяких системах керування базами даних відбувається блокування таблиць.

Отже, етап редизайну передбачає оптимізаційні зміни лише у рамках одного сховища даних. Але іноді внесених змін виявляється недостатньо для досягнення бажаних показників продуктивності. Це може відбутися за умови існування не властивої поточній базі структури організації даних, або відбувалася поступова та непомітна трансформація до такої структури шляхом внесення змін відповідно до вимог замовника. Для цього існують два методи реінжинірингу. Один із них – міграція даних. Хоча цей процес може бути громіздким і трудомістким, вважається, що переваги значно переважують кінцеві витрати [4]. Міграція також корисна, якщо на етапі редизайну не вдалося досягти бажаних показників. У публікації [5] наведено приклад методу реінжинірингу шляхом міграції даних з реляційного сховища до графової бази даних. Перевагами такого підходу є підвищення продуктивності, а недоліком – ускладнення підтримки та обслуговування нової СКБД, оскільки нова технологія вимагає специфічних навичок для якісної роботи.

Однак існують випадки, коли у системі дані різнорідної структури зберігаються у сховищі одного типу і виникає необхідність декомпозиції бази даних на декілька гетерогенних сховищ та подальшої міграції. Часто гетерогенні данні пов'язані між собою, а тому потребують шару абстракції для комунікації між собою. Це може бути зв'язувальне програмне забезпечення, що складається із обгортки, яка у свою чергу виконує функції формування запитів до гетерогенних сховищ. У статті [6] описано метод використання різнорідних баз даних у рамках реінжинірингу та дозволяє виконувати вибірку із декількох сховищ. Проте, підтримка таких систем може виявитись не простим завданням, оскільки масштабування сховищ даних до нових типів вимагає значних витрат та ресурсів.

Отже, найскладнішим етапом реінжинірингу є декомпозиція інформаційного сховища та міграція даних. З іншого боку, досить корисним методом редизайну є застосування матеріалізації у базах даних, що означає зберігання копії результату виконання певного запиту довільної складності. Матеріалізований запит будь-якої алгоритмічної складності при доступі до представлення матиме складність, що рівна складності простої вибірки. Це $O(1)$, якщо застосовано hash-індекс, $O(\log(n))$, якщо використано – b-tree індекс, або $O(n)$ – повне сканування за відсутності індексів. Як наслідок, зменшується навантаження на процесор і диск [7]. Проте недоліком матеріалізованих представлень є їх статичність, тобто, сама їх концепція полягає у тому, щоб зберігати великі набори даних уже у тому вигляді, у якому необхідно їх отримати. При синхронізації представлень та вихідної таблиці у деяких СКБД відбувається блокування таблиці, а час блокування залежить від тривалості оновлення представлення, що в свою чергу залежить від складності запити та об'єму даних у таблиці. Саме тому їх зазвичай використовують для отримання даних із рідко оновлюваних таблиць, а отже, потреба у синхронізації виникає лише один або декілька разів у певний проміжок часу [8].

Одним з методів збільшення швидкості оновлення матеріалізованих представлень є метод інкрементного оновлення. При перезаписі враховуються лише зміни у вихідних таблицях. Часто для цього використовуються різні структури даних, як, наприклад, LSM-дерево, яке є добре відомою структурою, прийнятою для покращення продуктивності запису [9]. Перевагами інкрементного оновлення є краща продуктивність перезапису матеріалізованих представлень за рахунок вибіркового застосування змін. Проте, не усі реляційні системи підтримують такий підхід. Крім того, процес оновлення матеріалізованих представлень блокує таблиці для запису, що може негативно вплинути на швидкість вставки та оновлення.

Метод відкладеного оновлення ґрунтується на перезаписі представлень один раз у деякий проміжок часу. Крім того, основним завданням є підтримка актуальності при вибірці, а це означає введення додаткових сховищ або таблиць у систему, що відповідатимуть за кешування вхідних даних. Але більшість авторів, які пропонують різні методи підтримки представлень, не наводять конкретних реалізацій [10]. Багато реляційних СКБД на сьогодні досі не реалізують ні один із наявних методів підтримки представлень.

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

Слід зазначити, що потенціал матеріалізованих представлень може бути ефективно використаний при роботі із гетерогенними даними. Таким чином проведення подальшого етапу реінжинірингу шляхом декомпозиції або міграції неоднорідної інформації виявиться надлишковим, а матеріалізовані представлення потенційно можуть вирішити проблему продуктивності на рівні редизайну сховища.

Формулювання цілей статті

Метою роботи є удосконалення методу матеріалізованих представлень, що дасть змогу використовувати усі переваги матеріалізації даних у комбінації із реалізованим на основі принципів кешування, алгоритмом синхронізації змін, а отже, підвищить показники продуктивності програмної системи та значно знизить вартість підтримки та супроводу апаратного та програмного забезпечення за рахунок досягнення ефективних результатів на етапі редизайну схеми та відмови від проведення декомпозиції сховища даних, як наступного етапу реінжинірингу бази.

Виклад основного матеріалу

Тому запропоновано удосконалення методу матеріалізованих представлень за рахунок застосування принципів стратегій кешування Write Back та Read Though у проектуванні табличної структури. Це дозволить використовувати матеріалізовані представлення на постійній основі, а також підвищить продуктивність програмної системи. За допомогою такого підходу реінжиніринг бази даних можна завершити на етапі редизайну. Для реалізації методу необхідно спроектувати спеціальну табличну структуру. Спрощена схема подана на рис. 1.

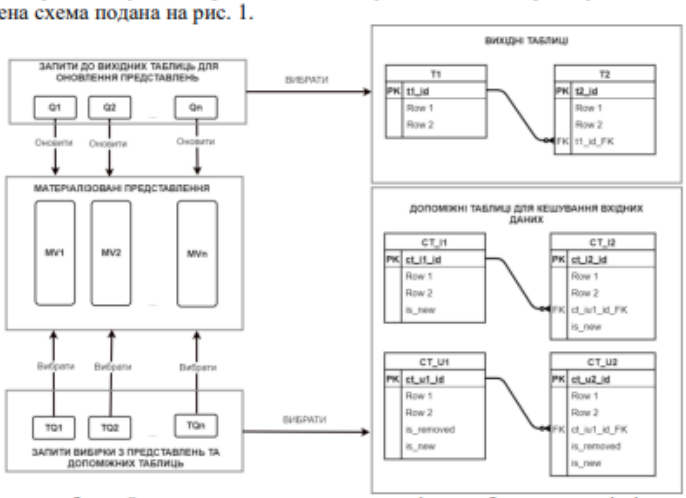


Рис. 1. Загальна схема табличної структури з використанням допоміжних таблиць та матеріалізованих представлень

Згідно рис. 1, таблиці T1 та T2 є вихідними. CT_I1 та CT_I2 – є копіями вихідних, що зберігатимуть дані для вставки, а CT_U1 та CT_U2 – для оновлення даних. Допоміжні таблиці кешуватимуть зміни та при синхронізації застосовуватимуть їх до вихідних таблиць.

MV1, MV2, …, MVn – матеріалізовані представлення, що зберігають результат виконання запитів Q1, Q2, …, Qn відповідно. Вибірка даних буде відбуватись з урахуванням допоміжних таблиць шляхом їх об'єднання з представленням. Дані, що помічені як is_removed не будуть враховуватись у вибірці з представлень, оскільки вони при синхронізації мають бути видалені з вихідних таблиць. У разі оновлення, записи із відповідним ідентифікатором у мат. представленнях не враховуватимуться при вибірці, оскільки в такому випадку поступає нова інформація, що зумовлює використання даних лише із допоміжних таблиць. Для синхронізації змін вибиратиметься проміжок часу, протягом якого необхідно здійснити оновлення. Найчастіше встановлюється час найменшого навантаження на сервер. Алгоритм подано на рисунку 2.

Асимптотична складність вибірки з представлення завжди рівна складності звичайної вибірки. Нехай, є дві вихідні таблиці із кількістю даних N_1 та N_2 відповідно. При цьому, нехай, вважатимемо, що у вихідних таблицях існує досить велика кількість записів, а тому $N_1 \gg 100, N_2 \gg 100$.

Зазвичай для прискорення вибірки із таблиць з великою кількістю даних встановлюють індекси на зовнішні ключі, що дозволяє виконувати вибірку за лінійний час, уникаючи вкладених запитів.

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

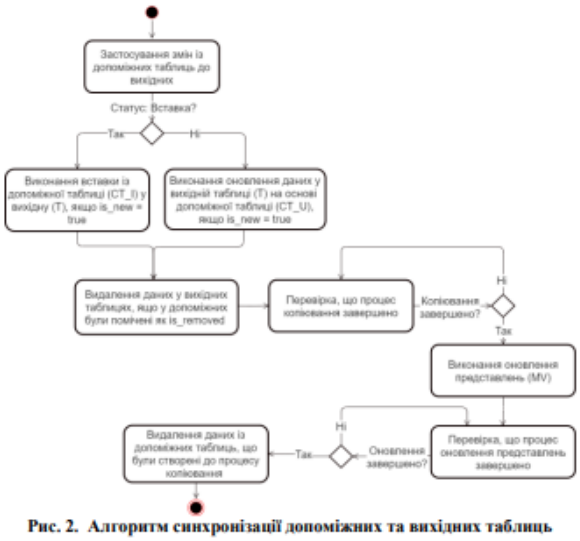


Рис. 2. Алгоритм синхронізації допоміжних та вихідних таблиць

Асимптотична складність алгоритму вибірки у кращому випадку при застосуванні техніки Sort-Merge за умови унікальності даних у двох таблицях становитиме:

$$O(N_1 + N_2), \tag{1}$$

де O – нотація Ландау;

N_1, N_2 – обсяг першої та другої вихідної таблиці відповідно [11].

Нехай у систему введено матеріалізовані представлення із кількістю даних M та тимчасові таблиці для вставки із обсягами I_1 та I_2. При цьому згідно алгоритму синхронізації, дані із цих таблиць періодично видаляються, а отже, максимальна кількість даних буде значно меншою за кількість у вихідних таблицях, тобто:

$I_1 \ll N_1, I_2 \ll N_2, N \gg 100$. При цьому, згідно з публікацією [12], асимптотична складність вибірки у кращому випадку при використанні b-tree індексу становитиме $O(log_2(M))$. Для вибірки даних об'єднуватимемо матеріалізоване представлення і допоміжні таблиці. Загальна асимптотична складність алгоритму у кращому випадку, взявши до уваги (1), становитиме:

$$B = O(log_2(M) + I_1 + I_2), \tag{2}$$

де B – асимптотична складність алгоритму вибірки із представлення та допоміжних таблиць для вставки;

M – кількість записів у матеріалізованому представленні;

I_1, I_2 – кількість записів у першій та другій допоміжній таблиці для вставки відповідно.

Проте існують випадки, коли записи у одну із таблиць не виконуються, а використовуються існуючі ключі. Може виникнути ситуація, при якій нові дані записані у одну з таблиць, але зовнішній ключ вказує на уже існуючі дані, а отже, у іншій зв'язаній допоміжній таблиці нового запису не буде і доведеться виконувати вибірку із вихідної таблиці. Час виконання зросте, проте, для того, щоб уникнути подальших звернень до вихідних таблиць, дані, що вперше отримані із неї необхідно кешувати шляхом перенесення у допоміжну таблицю і подальші звернення використовуватимуть дані у ній. При цьому такі записи необхідно помітити, щоб уникнути їх дублювання при перенесенні у вихідну таблицю (стовбець is_new на рис. 1).

Додамо таблиці, що будуть містити зміни, які у вихідних таблицях необхідно застосувати для оновлення записів: U_1 та U_2. Оскільки при вибірці необхідно враховувати лише дані з таблиць для оновлення і не враховувати дані на видалення, асимптотична складність алгоритму збільшиться, проте це не суттєво для невеликих об'ємів даних, оскільки, $U_1 \ll N_1$ та $U_2 \ll N_2$. Взявши до уваги B з (2), отримаємо таку складність алгоритму у кращому випадку:

$$B + O(U_1 + U_2 + X), \tag{3}$$

Міжнародний науково-технічний журнал
«Вимірювальна та обчислювальна техніка в технологічних процесах»
ISSN 2219-9365

де U_1, U_2 – кількість записів у першій та другій допоміжній таблиці для оновлення відповідно;
X – додаткова складність, яка може виникнути при відсіюванні застарілих або видалених даних.

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Загалом було проаналізовано основні методи та етапи реінжинірингу бази даних, встановлено їх переваги та недоліки, детально досліджено матеріалізацію даних, як один із методів реінжинірингу інформаційного сховища та спосіб збільшення показників продуктивності бази даних. Запропоноване удосконалення методу матеріалізованих представлень дозволяє підвищити продуктивність виконання запитів вибірки будь-якої складності. Подальші дослідження спрямовані на підвищення продуктивності вибірки даних із допоміжних таблиць шляхом удосконалення методів фільтрації застарілих та видалених даних, а також удосконалення процесів синхронізації даних між допоміжними та вихідними таблицями.

Література

1. N. Kumari. SQL Server Query Optimization Techniques - Tips for Writing Efficient and Faster Queries / Kumari N. // International Journal of Scientific and Research Publications. – Punjab, India, June 2012. – P. 1.

2. E. Morelli. Autonomous Re-indexing / Morelli E., Almeida A., Lifschitz S., Maria Monteiro J., Machado J. // Proceedings of the 27th Annual ACM Symposium on Applied Computing. – USA, March 2012. – P. 893–896.

3. J. Goldstein. Optimizing Queries Using Materialized Views: A practical, scalable solution / Goldstein J., Larson P.-Å. // Proceedings of the 2001 ACM SIGMOD international conference on Management of data. – June 2001. – DOI: 10.1145/375663.375706.

4. F. Oladipo. Re-engineering legacy data migration methodologies in critical sensitive systems / Oladipo F., Raiyetumbi J. // Journal of Global Research in Computer Science . – Kogi, Nigeria, November 2015.

5. Y. Unal. Migration of Data from Relational Database to Graph Database / Unal Y., Oguztuzun H. // International Conference on Information and Software Technologies. – Istanbul, Turkey, March 2018. – DOI: 10.1145/3200842.3200852.

5. H. Zhang. Unified SQL Query Middleware for Heterogeneous Databases / Zhang H. // Journal of Physics: Conference Series. – Shanghai, China, 2021.

6. A. Gupta. Problems, Techniques, and Applications / Gupta A., Mumick I. // IEEE Data Eng. Bull. – P. 1.

7. V. Kumar. Materialized View Selection Using Iterative Improvement / Kumar V., Kumar S. // Advances in Intelligent Systems and Computing. – New Delhi, India, 2013. – DOI: 10.1007/978-3-319-91458-9_42

8. H. Duan. Incremental Materialized View Maintenance on Distributed Log-Structured Merge-Tree / Duan H., Hu H., Qian W., Ma H., Wang X., Zhou A. // Database Systems for Advanced Applications. – May 2018.

9. A. Gosaina. Architecture Based Materialized View Evolution: A Review / Gosaina A., Sabharwal S., Gupta R. // Procedia Computer Science. – 2015. – P. 257. – DOI: 10.1016/j.procs.2015.04.179.

10. M.-C. Albutiu. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems / Albutiu M.-C., Kemper A., Neuman T. // Proceedings of the VLDB Endowment. – Germany, June 2012.

11. D. Comer. Computing Surveys: The Ubiquitous B-Tree / Comer D. // ACM Computing Surveys. – West Lafayette, Indiana, USA, June 1979. – P. 127.

References

1. N. Kumari. SQL Server Query Optimization Techniques - Tips for Writing Efficient and Faster Queries / Kumari N. // International Journal of Scientific and Research Publications. – Punjab, India, June 2012. – P. 1.

2. E. Morelli. Autonomous Re-indexing / Morelli E., Almeida A., Lifschitz S., Maria Monteiro J., Machado J. // Proceedings of the 27th Annual ACM Symposium on Applied Computing. – USA, March 2012. – P. 893–896.

3. J. Goldstein. Optimizing Queries Using Materialized Views: A practical, scalable solution / Goldstein J., Larson P.-Å. // Proceedings of the 2001 ACM SIGMOD international conference on Management of data. – June 2001. – DOI: 10.1145/375663.375706.

4. F. Oladipo. Re-engineering legacy data migration methodologies in critical sensitive systems / Oladipo F., Raiyetumbi J. // Journal of Global Research in Computer Science . – Kogi, Nigeria, November 2015.

5. Y. Unal. Migration of Data from Relational Database to Graph Database / Unal Y., Oguztuzun H. // International Conference on Information and Software Technologies. – Istanbul, Turkey, March 2018. – DOI: 10.1145/3200842.3200852.

6. H. Zhang. Unified SQL Query Middleware for Heterogeneous Databases / Zhang H. // Journal of Physics: Conference Series. – Shanghai, China, 2021.

7. A. Gupta. Problems, Techniques, and Applications / Gupta A., Mumick I. // IEEE Data Eng. Bull. – P. 1.

8. V. Kumar. Materialized View Selection Using Iterative Improvement / Kumar V., Kumar S. // Advances in Intelligent Systems and Computing. – New Delhi, India, 2013. – DOI: 10.1007/978-3-319-91458-9_42

9. H. Duan. Incremental Materialized View Maintenance on Distributed Log-Structured Merge-Tree / Duan H., Hu H., Qian W., Ma H., Wang X., Zhou A. // Database Systems for Advanced Applications. – May 2018.

10. A. Gosaina. Architecture Based Materialized View Evolution: A Review / Gosaina A., Sabharwal S., Gupta R. // Procedia Computer Science. – 2015. – P. 257. – DOI: 10.1016/j.procs.2015.04.179.

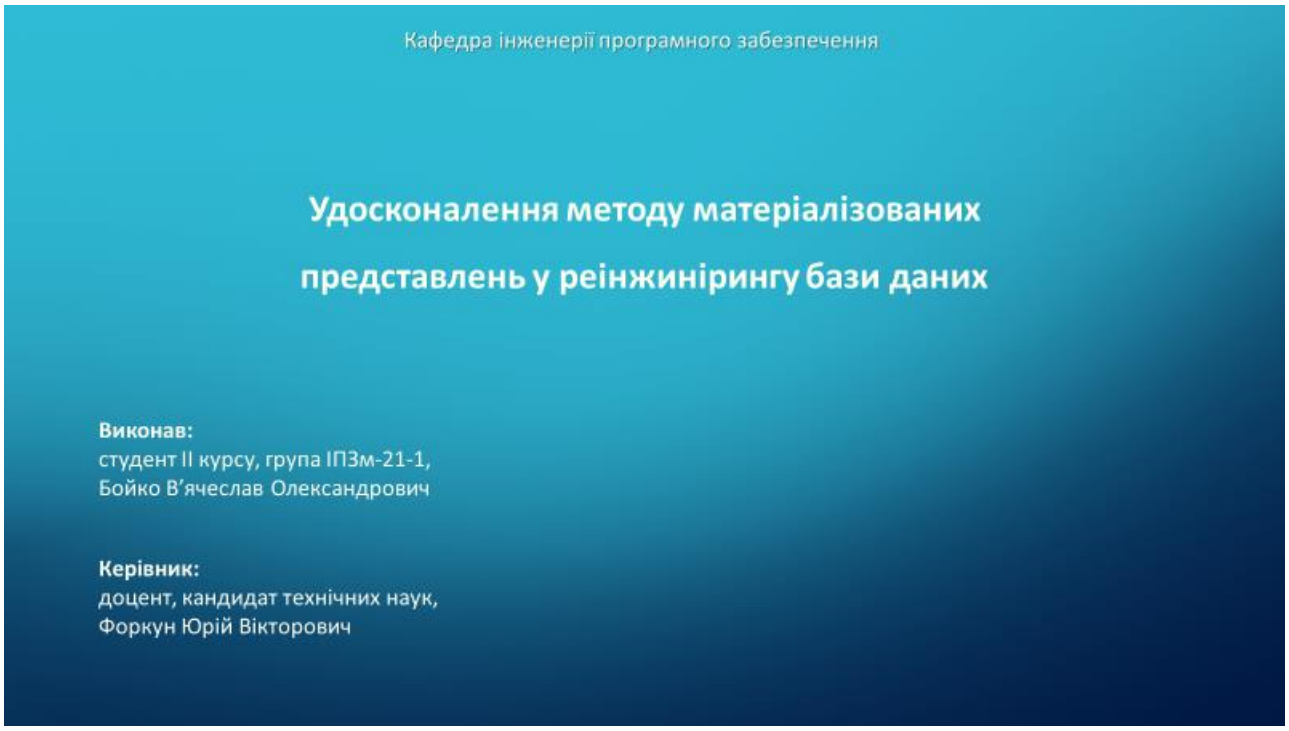
11. M.-C. Albutiu. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems / Albutiu M.-C., Kemper A., Neuman T. // Proceedings of the VLDB Endowment. – Germany, June 2012.

12. D. Comer. Computing Surveys: The Ubiquitous B-Tree / Comer D. // ACM Computing Surveys. – West Lafayette, Indiana, USA, June 1979. – P. 127.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ



Об'єкт, предмет і мета дослідження

- Об'єкт дослідження – реінжиніринг бази даних.
- Предмет дослідження – методи та способи реінжинірингу бази даних.
- Мета дослідження – удосконалення методу матеріалізованих представлень, що дасть змогу використовувати усі переваги матеріалізації даних у комбінації із реалізованим на основі принципів кешування, алгоритмом синхронізації змін, а отже, підвищить показники продуктивності програмної системи та значно знизить вартість підтримки та супроводу апаратного та програмного забезпечення за рахунок досягнення ефективних результатів на етапі редизайну схеми та відмови від проведення декомпозиції сховища даних, як наступного етапу реінжинірингу.

Актуальність теми

- Актуальність роботи полягає у тому, що основним впливовим артефактом великої кількості програмних систем є сховища даних, оскільки характер та структура організації інформації має сильний вплив на продуктивність усієї програмної системи. Дослідження процесу реінжинірингу на практиці дає змогу встановити переваги та недоліки наявних методів та встановити необхідність у розробці нових, які будуть здатні підвищити продуктивність та зменшити витрати на підтримку різнорідного програмного та апаратного забезпечення.

Завдання дослідження

Відповідно до мети необхідно вирішити такі задачі дослідження:

- провести теоретичний аналіз сфери реінжинірингу баз даних;
- виконати аналіз та порівняння методів реінжинірингу бази даних у програмних системах;
- описати наявні механізми реалізації реінжинірингу, проаналізувати їх переваги та недоліки;
- дослідити роботу матеріалізованих представлень та реалізовані на їх основі алгоритми, що підвищують продуктивність запитів вибірки;
- удосконалити метод матеріалізованих представлень у рамках реінжинірингу бази даних;
- провести тестування та практичну апробацію отриманих результатів, дослідити ефективність запропонованих рішень;
- проаналізувати отримані результати та сформувати рекомендації щодо доцільності впровадження результатів дослідження.

Аналіз галузі та виділення проблем

Основні проблеми у галузі реінжинірингу баз даних:

- при зберіганні даних різнорідної структури у сховищі одного типу виникає необхідність декомпозиції бази даних на декілька гетерогенних сховищ, що ускладнює процеси підтримки такої системи та підвищує витрати ресурсів на супровід апаратного та програмного забезпечення;
- практично неможливо оптимізувати складні запити у реляційних сховищах, що містять різнорідну структуру впорядкування даних, а реплікація даних лише ненадовго вирішить проблему;
- взаємодія між гетерогенними сховищами вимагає реалізації проміжного програмного забезпечення або використання наявних гібридних систем керування базами даних, що значно збільшить час та витрати на розробку програмного продукту;
- матеріалізація даних значно прискорює виконання запитів вибірки, проте працюватиме лише для даних, які не потребують частого оновлення;

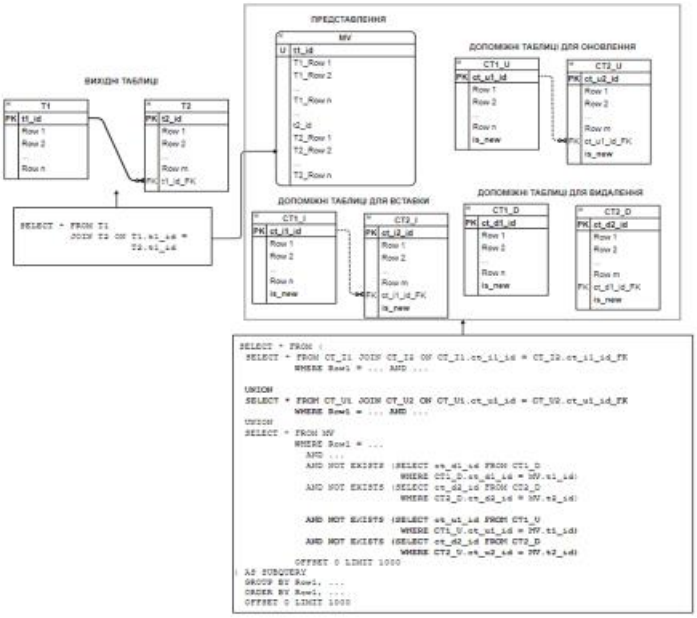
Удосконалений метод матеріалізованих представлень

Метод матеріалізації даних використовується з метою збереження результатів виконання запитів у окремих об'єктах бази даних – матеріалізованих представленнях для підвищення продуктивності складних запитів. Основний недолік таких представлень – необхідність їх оновлення, що означає повторне виконання складного запиту. Тому вони використовуються у випадках, коли не потрібно постійне оновлення та вибірка нових даних.

У дипломній роботі запропоновано удосконалення методу матеріалізованих представлень шляхом додавання допоміжних таблиць, у які записуватимуться нові зміни, а саме оновлення представлень відбуватиметься інтервально. При цьому дані з допоміжних таблиць перед оновленням будуть видалятись та застосовуватись у вихідних таблицях.

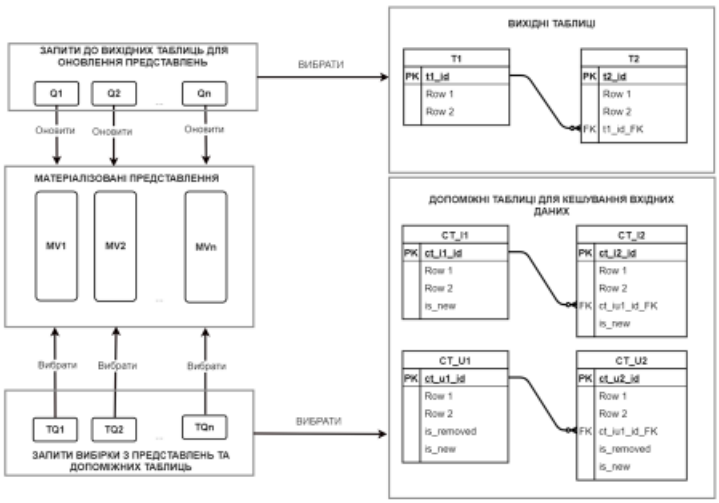
Удосконалений метод матеріалізованих представлень

Таблична структура удосконаленого методу матеріалізованих представлення прикладі двох зв'язаних таблиць



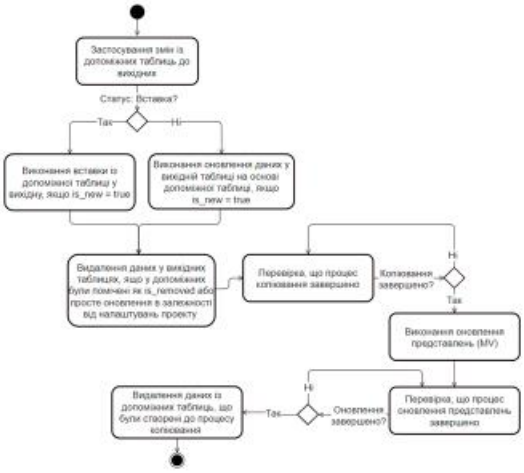
Удосконалений метод матеріалізованих представлень

Узагальнена таблична структура удосконаленого методу матеріалізованих представлень



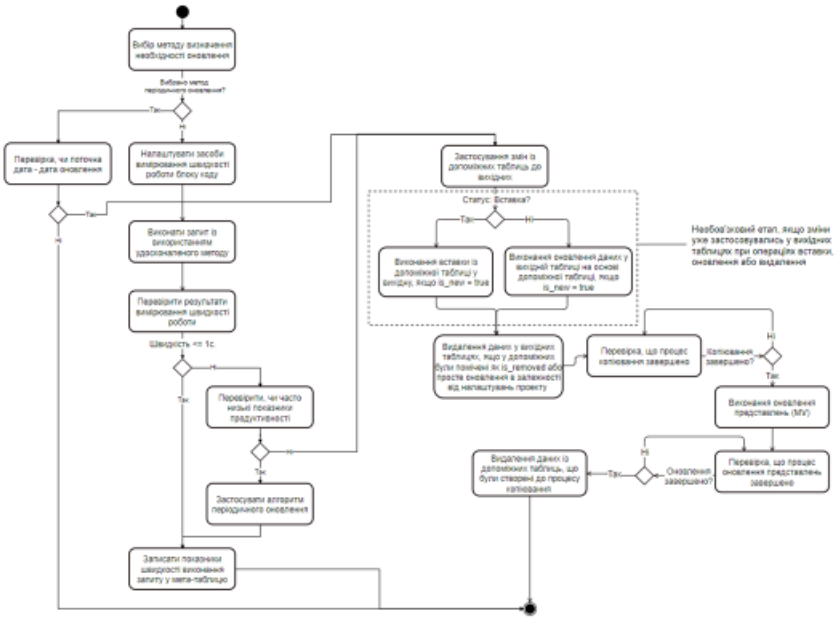
Удосконалений метод матеріалізованих представлень

Алгоритм синхронізації допоміжних та вихідних таблиць



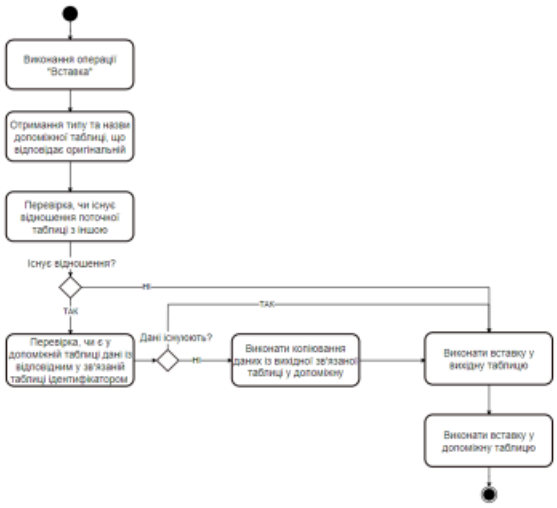
Удосконалений метод матеріалізованих представлень

Розширений алгоритм синхронізації допоміжних та вихідних таблиць



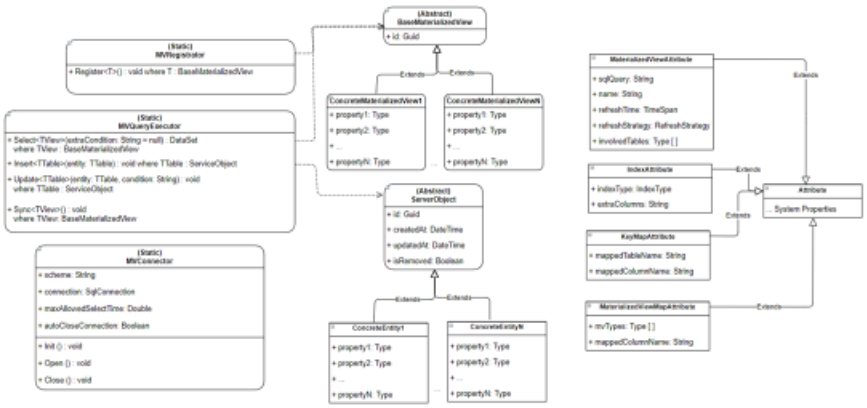
Удосконалений метод матеріалізованих представлень

Алгоритм вставки



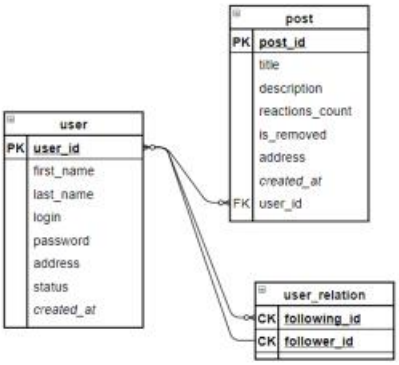
Удосконалений метод матеріалізованих представлень

Діаграма класів бібліотеки Advanced Materialization



Апробація результатів

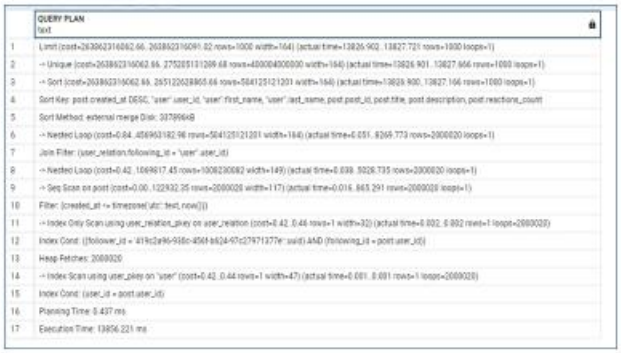
Для проведення апробації було спроектовано частину реляційної бази даних, що використовується у корпоративному додатку для пошуку речей. Для цього було створено три основні таблиці: user, post, user_relation та заповнено їх великою кількістю згенерованих даних (100 тисяч – 10 мільйонів). Для підвищення продуктивності вибірки, на стовпці, що використовуються для сортування було встановлено додаткові b-tree індекси.



Апробація результатів

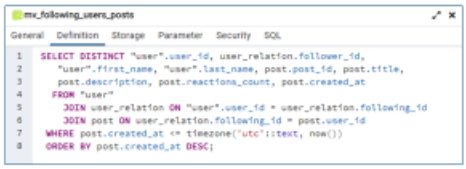
Для тестування продуктивності було обрано складний тестовий запит: **вибрати 1000 публікацій різних підписників користувача з ідентифікатором '419c2a96-930c-456f-b624-97c27971377e'**. За робочу СКБД взято PostgreSQL.

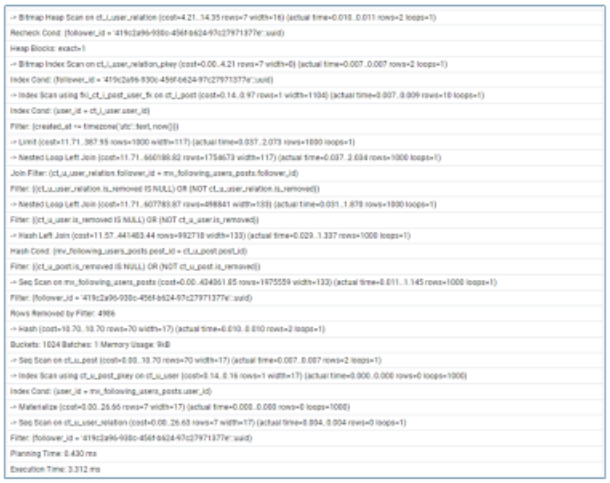
```
SELECT DISTINCT "user".user_id,
    user_relation.follower_id,
    "user".first_name,
    "user".last_name,
    post.post_id,
    post.title,
    post.description,
    post.reactions_count,
    post.created_at

  FROM "user"
    JOIN user_relation ON "user".user_id = user_relation.following_id
    JOIN post ON user_relation.following_id = post.user_id
    WHERE post.created_at <= timezone('utc'::text, now()) and
          follower_id = '419c2a96-930c-456f-b624-97c27971377e'
    ORDER BY post.created_at DESC
    OFFSET 0 LIMIT 1000
```



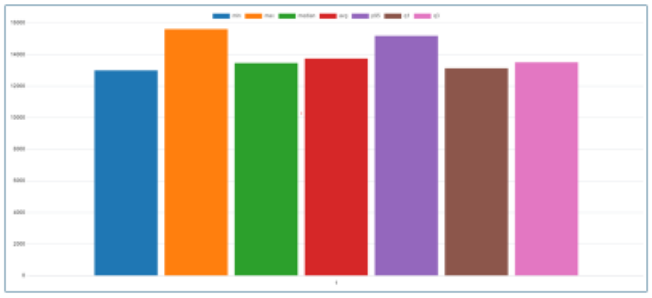
Апробація результатів

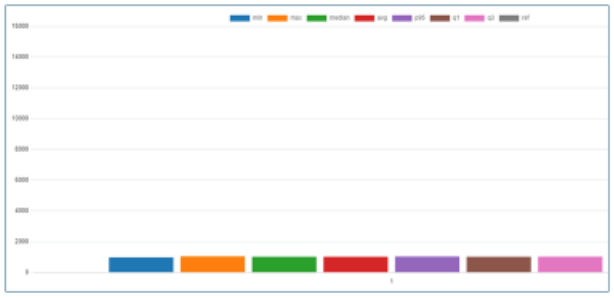
Використовуючи удосконалений метод матеріалізованих представлень, у схему введені додаткові таблиці та матеріалізоване представлення, що зберігає результати запиту вибірки публікацій.

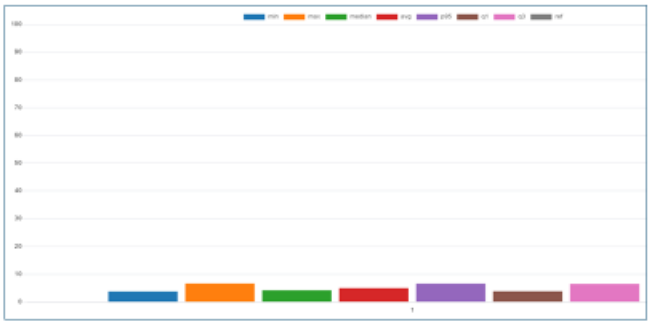




Апробація результатів







Отримані результати

Удосконалений метод матеріалізованих представлень дозволяє:

- підвищити швидкість роботи із даними різнорідної структури та збільшує продуктивність виконання складних запитів вибірки;
- зменшити витрати на облаштування додаткових апаратних та програмних засобів за рахунок уникнення проведення декомпозиції сховища;

Наукова новизна

- Отримав подальший розвиток метод матеріалізованих представлень, що дозволило підвищити продуктивність запитів вибірки із бази даних та відмовитися від етапу декомпозиції сховища. Застосовані техніки кешування даних за допомогою допоміжних таблиць дозволили використання матеріалізації даних на постійній основі, а також оновлення матеріалізованих представлень лише у певний період часу, що значно підвищує продуктивність програмного забезпечення.

Практична цінність

Практична цінність отриманих результатів полягає в успішному удосконаленні методу матеріалізованих представлень у рамках реінжинірингу бази даних. Завдяки поліпшеним характеристикам продуктивності у порівнянні з класичними рішеннями, удосконалений метод може бути успішно використаний при проектуванні гетерогенних систем, а програмне забезпечення, що буде його використовувати, матиме нижчу вартість підтримки та обслуговування і стабільно високу продуктивність.

Наукові публікації

1. Опублікована одна стаття у фаховому науковому виданні «Вісник Хмельницького національного університету. Серія «Вимірювальна та обчислювальна техніка в технологічних процесах»:

Бойко В.О., Форкун Ю.В. Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних // Вісник Хмельницького національного університету. – 2022. – № 3 (71) – С. 87-91 .


Дякую за увагу!

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Бойка В. О.
факультет ІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

02.12.22
дата

підпис

Mon Dec 05 11:10:10 EET 2022, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 14.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

	Документ		Сумарний збіг по Базі Даних	
ID: 108929 Назва: КРМ на тему: "Удосконалення методу матеріалізованихпредставлень у реінжинірингу бази даних" Додано в БД: 2022-12-05 Автора: Бойко В.О. Керівники: Форкун Ю.В. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	128343	980	22332 (17%)	190 (19%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
107858	Назва: Удосконалення методів та засобів реінжинірингу корпоративних програмних систем Додано в БД: 2022-10-04 Автора: В. О. Бойко Керівники: Форкун Ю.В. Консультанти: Опоненти:	18440 (14.0%)	147 (15.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013146129

Дата перевірки:
02.12.2022 10:25:56 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
02.12.2022 10:29:26 EET

ID користувача:
100005589

Назва документа: **Записка Бойко В.О.без дод**

Кількість сторінок: **92** Кількість слів: **18230** Кількість символів: **145075** Розмір файлу: **1.68 MB** ID файлу: **1012913624**

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

5.53%
Схожість

Найбільша схожість: **4.09%** з джерелом з Бібліотеки (ID файлу: **1009383373**)

1.04% Джерела з Інтернету	104	Сторінка 94
5.01% Джерела з Бібліотеки	155	Сторінка 95

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Підозріле форматування	16 сторінок

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

освітнього ступеня «магістр»

Магістр Бойко В'ячеслав Олександрович

Тема Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних

Спеціальність 121 « Інженерія програмного забезпечення»

Обсяг дипломної роботи:

Кількість сторінок дипломної роботи 123 .

1. Короткий зміст роботи та прийнятих рішень У процесі дипломного проектування досліджено галузь реінжинірингу баз даних та сучасні методи і способи реінжинірингу, визначено невирішені проблеми у галузі, на основі чого удосконалено метод матеріалізованих представлень, виконано його програмну реалізацію у вигляді бібліотеки, навантажувальне тестування сховища даних та модульне тестування реалізованого плагіну. Нерозв'язані задачі запропоновано вирішити шляхом удосконалення методу матеріалізованих представлень, що полягає у використанні допоміжних таблиць для кешування вхідних змін та подальше їх застосування при синхронізації.

2. Висновок про відповідність роботи дипломному завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині

2. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обґрунтовується актуальність теми роботи, формулюються цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі дипломної роботи досліджено та проаналізовано сферу реінжинірингу сховищ даних, останні публікації, наукові статті та джерела, існуючі методи та способи проведення реінжинірингу бази даних з метою виявлення невирішених проблем. У другому розділі роботи проаналізовано моделі, концепції та алгоритми вирішення проблем реінжинірингу баз даних. Нерозв'язані задачі запропоновано вирішити шляхом удосконалення методу матеріалізованих представлень, що полягає у використанні допоміжних таблиць для кешування вхідних змін та подальше їх застосування при синхронізації. У третьому розділі описана технологія реалізації удосконаленого методу матеріалізованих представлень. У четвертому розділі описано програмну реалізацію, навантажувальне та модульне тестування розроблюваного плагіну на основі удосконаленого методу реінжинірингу. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленого програмного засобу та його функціональної придатності. Обґрунтована ефективність створених методів та засобів та розроблено рекомендації з їх застосування при реінжинірингу сховища даних.

4. Позитивні сторони роботи Дипломна робота містить інноваційні рішення, зокрема, було доведено доцільність використання матеріалізованих представлень на постійній основі, удосконалено метод та, що дозволило оптимізувати програмні системи за рахунок підвищення показників продуктивності запитів вибірки підтримання, а

реалізована бібліотека дозволяє автоматизувати процес створення матеріалізованих представлень, додаткових таблиць та операції над ними.

5. Негативні сторони роботи Алгоритм, що реалізовано у роботі потребує додаткового аналізу, оскільки у подальшому необхідно встановити, при якій кількості даних у допоміжних таблицях швидкість значно знизиться. Крім того, необхідно віднайти рішення задачі блокування таблиць бази даних у період оновлення матеріалізованих представлень та синхронізації змін

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8.Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)

професор кафедри комп'ютерної інженерії та інформаційних систем ХНУ, Сергій Лисенко

«30» листопада 2022 р.

(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Удосконалення методу матеріалізованих представлень у реінжинірингу бази даних»
Автор: Бойко В'ячеслав Олександрович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Форкун Юрій Вікторович, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає **5,53%** і адресується до 104 джерел з Інтернету та 155 джерел з бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ Ю. В. Форкун

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк