Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів міграції даних
з реляційної в графову базу даних
(тема)

Виконав:
Здобувач 2 року навчання
групи ІПЗм-23-4

Дмитро МИХНЕВИЧ
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Оксана МАЗУРОВА
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ проф. Кирило СМЕЛЯКОВ
(підпис)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)
Тип програми освітньо-наукова програма
Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___»_____ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Михневичу Дмитру Костянтиновичу
(прізвище, ім’я, по батькові)

1. Тема роботи

Дослідження методів міграції даних з реляційної в графову базу даних.

Затверджена наказом по університету від 15.04.2025 р №290 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 16.06.2025

3. Вихідні дані до роботи:

Електронні ресурси за обраною тематикою, методи міграції даних з реляційної в графову БД, бази даних Neo4j, Microsoft SQL Server, середовище розробки Visual Studio 2022, мови SQL, Cypher, C#.

4. Перелік питань, що потрібно опрацювати в роботі:

аналіз предметної та проблемної області, постановка задачі, аналіз методів міграції реляційних даних, розробка логічних моделей, планування експериментального дослідження, проведення експериментів та аналіз отриманих результатів.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	27.01.25 – 30.01.25	виконано
2	Розробка постановки задачі	30.01.25 – 01.02.25	виконано
3	Аналіз та вибір реляційних та графових СУБД	01.02.25 – 03.02.25	виконано
4	Теоретичне дослідження	03.02.25 – 09.02.25	виконано
5	Планування експериментального дослідження	09.02.25 – 11.02.25	виконано
6	Аналіз та моделювання предметної області	11.02.25 – 15.02.25	виконано
7	Розробка запитів для експериментального дослідження	15.02.25 – 20.02.25	виконано
8	Проектування та розробка ПЗ	20.02.25 – 01.03.25	виконано
9	Проведення експериментів	01.03.25 – 01.04.25	виконано
10	Підготовка статті	01.04.25 – 20.05.25	
11	Підготовка пояснювальної записки	10.05.25 – 25.05.25	виконано
12	Підготовка презентації та доповіді	25.05.25 – 31.05.25	виконано
13	Перевірка на плагіат	01.06.25	
14	Нормоконтроль	05.06.25	виконано
15	Рецензування	08.06.25	виконано
16	Занесення диплома в електронний архів	11.06.25	виконано
17	Попередній захист	12.06.25	виконано
18	Допуск до захисту у зав. кафедри	14.06.25	виконано

Дата видачі завдання 27 . січня .2025р.

Студент _____ Дмитро МИХНЕВИЧ
(підпис)

Керівник роботи доц. Оксана МАЗУРОВА
(посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка містить: 114 с., 21 рис., 9 табл., 19 джерел.

АЛГОРИТМ МІГРАЦІЇ, БАЗА ДАНИХ, ВЕРШИНА, ГРАФОВА МОДЕЛЬ ДАНИХ, РЕБРО, РЕЛЯЦІЙНА МОДЕЛЬ ДАНИХ, СУТНІСТЬ, ТАБЛИЦЯ, MSSQL, NEO4J.

Об’єктом дослідження є графові моделі даних в інформаційних системах.

Метою роботи є проведення дослідження існуючих підходів до міграції даних з реляційної в графову БД з подальшим удосконаленням розглянутих методів міграції.

Методами розробки та проєктування є аналіз проблемної області дослідження, метод багатокритеріального прийняття рішень для вибору графової СУБД, методи логічного проєктування баз даних.

В результаті роботи були досліджені методи міграції даних з реляційної в графову БД, запропоновано шляхи удосконалення досліджуваних методів, розроблені реляційна та графові логічні моделі, сплановане експериментальне дослідження, розроблено програмне забезпечення та запити для проведення експериментів, проведено експерименти та проаналізовано їх результати.

MIGRATION ALGORITHM, DATABASE, VERTEX, GRAPH DATA MODEL, EDGE, RELATIONAL DATA MODEL, ENTITY, TABLE, MSSQL, NEO4J.

The object of research is graph data models in information systems.

The purpose of the work is to conduct a study of existing approaches to data migration from a relational to a graph database with subsequent improvement of the migration methods considered.

The development and design methods are the analysis of the problem area of the study, a multi-criteria decision-making method for choosing a graph DBMS, and methods of logical database design.

As a result of the work, the methods of data migration from a relational to a graph database were studied, ways of improving the studied methods were proposed, relational and graph logical models were developed, an experimental study was planned, software and requests were developed for conducting experiments, experiments were conducted, and their results were analyzed.

Завідувачу кафедри
ПІ_____
(скорочена назва кафедри)
проф. Кирилу СМЕЛЯКОВУ_
(вчене звання, власне ім'я, прізвище)

ЗАЯВА

щодо самостійності виконання кваліфікаційної роботи та можливості її публікації(та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я,_____Михневич Дмитро Костянтинович_____
(прізвище, ім'я, по батькові)

здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи_____ІПЗм-23-4_____

кафедра_____програмної інженерії_____
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему

__«Дослідження методів міграції даних з реляційної в графову базу даних»___,
(назва роботи)

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата 30.05.25  Підпис

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій постійно зростають вимоги до ефективності зберігання, обробки та аналізу даних, адже кількість інформації, яка зберігається в мережі Інтернет, збільшується кожного дня. Протягом досить тривалого часу з задачею зберігання та обробки інформації добре справлялися реляційні бази даних, але, враховуючи сучасні тенденції збільшення інформації, реляційні БД все частіше відходять на другий план. Це пояснюється наявністю в реляційних базах даних певних обмежень при роботі з великою кількістю взаємопов’язаних даних.

Для вирішення цієї проблеми необхідно було розробити нові методи зберігання, аналізу та обробки складних структур даних. Так з’явилися NoSQL бази даних, в тому числі графові бази даних. Графові БД мають наступні переваги:

– оптимізовані для швидкого пошуку взаємопов’язаних сутностей;
– швидка адаптація до зміни схеми даних;
– підтримка ефективного горизонтального масштабування.

Деякі програмні системи, що були побудовані на основі реляційних БД, мали б кращу продуктивність, якби використовували саме графову базу даних для зберігання інформації.

Мета дослідження полягає в розробленні комплексного методологічного підходу до міграції даних з реляційної в графову БД, який забезпечує повноту та коректність перетворення вхідних структур.

Під час виконання магістерської кваліфікаційної роботи було проведено аналіз проблемної області, на основі якого було обрано цільову графову СУБД, а саме Neo4J, а також проведене логічне моделювання вхідних та вихідних структур, було спроєктовано та розроблене програмне забезпечення для проведення експериментів, розроблено та реалізовано запити для експериментального дослідження. Було проведено експериментальне дослідження, на основі його результатів було запропоновано рекомендації щодо

використання досліджуваних методів міграції даних з реляційної в графову базу даних.

Кваліфікаційна робота пройшла успішну перевірку на академічну доброчесність (див. додаток А).

На основі плану проведення дослідження та його результатів було розроблено презентацію (див. додаток Б).

За результатами роботи були опубліковані тези доповіді на сімнадцяту міжнародну науково-технічну конференцію «Інформаційні технології та автоматизація – 2024» (див. додаток В) та прийнято до опублікування статтю в журналі категорії Б (див. додаток Г).

Також, кваліфікаційна робота перевірена на відповідність вимогам оформлення (див. додаток Д).

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної галузі дослідження

Сучасний стан розвитку інформаційних технологій характеризується постійним зростанням обсягів та складності структурованих даних. Традиційні реляційні бази даних, незважаючи на їх тривалу історію використання, демонструють певні обмеження при роботі зі складнопов’язаними даними. Також реляційні БД мають певні проблеми з підтримкою масштабованості [1]. Це викликає необхідність пошуку альтернативних підходів до зберігання та обробки інформації.

Графові бази даних останнім часом набувають все більшої популярності через здатність ефективно представляти складні взаємозв'язки між сутностями. За даними звіту db-engines [2], графові бази даних входять до топ-10 систем управління базами даних, демонструючи стійку тенденцію до зростання популярності.

Графова база даних – це механізм зберігання даних, який поєднує базові графові структури вершин і ребер з технологією збереження та мовою запитів для створення бази даних, оптимізованої для зберігання та швидкого отримання щільно пов’язаних даних. На відміну від інших технологій баз даних, графові бази даних побудовані на концепції, що відносини між сутностями є такими ж важливими, як і самі сутності [3].

Також слід зазначити, що певна кількість програмних систем, що створені на основі реляційних СУБД, краще працювала б, якби зберігала дані в графовій БД [4]. Наприклад, проблема міграції даних з існуючих структурованих форматів у графову модель є актуальною для багатьох галузей, зокрема:

- соціальні мережі та аналіз соціальних зв'язків;
- системи рекомендацій;
- біоінформатика та кібербезпека;
- ігрова аналітика [5].

Загальна методологія до міграції даних складається з наступних етапів:

- аналіз вхідної структури даних;
- визначення правил відображення вхідної структури у результуючу структуру;
- генерація результуючої структури;
- верифікація отриманої структури.

Міграція даних з реляційних у графові бази даних може здійснюватися різними способами залежно від специфічних вимог проєкту та обмежень у часі. Важливо розрізняти основні типи міграції, кожен з яких має свої переваги та недоліки.

Холодна міграція передбачає повну зупинку роботи системи на період перенесення даних. Цей підхід забезпечує максимальну цілісність даних та дозволяє провести повну верифікацію результатів міграції. Однак холодна міграція неприйнятна для критично важливих систем, які повинні працювати безперервно. Час простою може становити від кількох годин до кількох діб залежно від обсягу даних, що переносяться.

Гаряча міграція, навпаки, дозволяє здійснювати перенесення даних без зупинки функціонування системи. Цей процес є значно складнішим з технічної точки зору, оскільки вимагає синхронізації змін, які відбуваються в системі під час міграції. Гаряча міграція часто реалізується поетапно, коли спочатку переносяться історичні дані, а потім налаштовується реплікація поточних змін.

Існує також змішаний підхід, який поєднує елементи обох типів міграції. На початковому етапі система працює в режимі паралельного ведення даних у двох базах, після чого на короткий час зупиняється для фінальної синхронізації та перемикання на нову систему.

Сьогодні існує декілька методів та інструментів для міграції даних з реляційних у графові бази даних. Кожен метод має свої особливості, переваги та обмеження, що ускладнює процес прийняття рішення для розробників та архітекторів систем.

Основна проблема полягає в тому, що ефективність різних методів міграції суттєво залежить від характеристик вхідних даних, структури реляційної бази,

цільової графової СУБД та специфічних вимог до продуктивності системи. Розробники часто стикаються з дилемою вибору між швидкістю міграції, повнотою збереження семантики даних та ефективністю подальшої роботи з графовою базою.

Відсутність комплексного порівняльного аналізу різних методів міграції ускладнює процес вибору оптимального рішення для конкретного проєкту.

Дослідження ефективності методів міграції даних потребують проведення комплексних експериментальних досліджень, які дозволяють об'єктивно оцінити переваги та недоліки різних підходів. Експериментальний підхід є особливо важливим у галузі баз даних, оскільки теоретичні оцінки часто не відображають реальної продуктивності системи в умовах конкретного навантаження.

Розробка експериментального дослідження в галузі баз даних традиційно включає кілька ключових етапів. Початковий етап передбачає детальний аналіз предметної області та вимог до системи. Цей аналіз дозволяє визначити основні сутності, їх атрибути та взаємозв'язки, які повинні бути враховані при проєктуванні бази даних.

Наступним етапом є створення інформаційно-логічної моделі даних, яка зазвичай представляється у вигляді ER-діаграми. ER моделювання дозволяє абстрагуватися від технічних деталей реалізації та зосередитися на семантиці предметної області. ER-модель відображає сутності, їх атрибути та різні типи зв'язків між ними.

Логічне моделювання передбачає перетворення інформаційно-логічної моделі у структуру, яка відповідає обраній моделі даних. Для реляційних баз даних це означає створення схеми таблиць з визначенням первинних та зовнішніх ключів. Для графових баз даних логічна модель описує типи вершин, їх властивості та типи ребер між ними.

Фізичне моделювання включає оптимізацію структури для конкретної СУБД з урахуванням особливостей зберігання даних, індексації та стратегій доступу. На цьому етапі приймаються рішення щодо розподілу даних, створення індексів та налаштування параметрів продуктивності.

Продуктивна база даних характеризується здатністю ефективно обробляти запити користувачів при збереженні прийнятного рівня використання системних ресурсів. Поняття продуктивності включає кілька взаємопов'язаних аспектів, які необхідно враховувати при оцінці ефективності системи.

Час відповіді є одним з найважливіших показників продуктивності, який характеризує інтервал між надсиланням запиту та отриманням результату. Для різних типів запитів прийнятний час відповіді може суттєво відрізнятися. Транзакційні системи зазвичай вимагають мілісекундного часу відповіді, тоді як аналітичні запити можуть виконуватися хвилинами.

Пропускна здатність системи визначається кількістю запитів, які система може обробити за одиницю часу. Цей показник особливо важливий для систем з високим навантаженням, де одночасно працює велика кількість користувачів. Пропускна здатність часто вимірюється в операціях за секунду або транзакціях за хвилину.

Використання ресурсів включає моніторинг завантаження процесора, обсягу використовуваної оперативної пам'яті, інтенсивності дискових операцій та мережевого трафіку. Ефективна система повинна забезпечувати високу продуктивність при раціональному використанні доступних ресурсів.

Масштабованість системи характеризує здатність підтримувати прийнятний рівень продуктивності при збільшенні обсягу даних або кількості користувачів. Графові бази даних часто демонструють кращу масштабованість для запитів, що включають обхід складних зв'язків, порівняно з реляційними системами.

Практична значущість дослідження полягає в удосконаленні існуючих алгоритмів міграції, що дозволить:

- мінімізувати втрати інформації;
- прискорити процес міграції даних;
- забезпечити семантичну цілісність перетворення.

Отже, дослідження методів міграції даних з реляційної в графову БД є актуальним.

1.2 Постановка задачі

Метою цієї роботи є дослідження та аналіз існуючих рішень для міграції даних з реляційної в графову базу даних, розробка нових підходів і методів для автоматичної міграції реляційних даних в графову БД, які б вирішували наявні проблеми та недоліки існуючих рішень.

В ході виконання роботи треба виконати наступні завдання:

а) провести аналіз існуючих графових СУБД та обрати відповідну СУБД для подальшого дослідження;

б) провести аналіз існуючих методів та алгоритмів міграції даних з реляційної у графову БД та розробити власний метод міграції в обрану графову СУБД;

в) провести планування експериментального дослідження методів міграції, а саме:

   1) провести аналіз та моделювання предметної області, яка містить складні структуровані дані, які доречно зберігати у вигляді графу;
   2) розробити моделі даних для вхідних структурованих даних;
   3) розробити алгоритми міграції даних з використанням різних методів;
   4) розробити критерії оцінки ефективності міграції та запити для перевірки коректності перетворення даних.

г) виконати реалізацію програмного забезпечення для проведення експериментального дослідження;

д) провести експериментальне дослідження методів;

е) розробити рекомендації щодо вибору та застосування методів міграції даних з реляційної у графову базу даних за результатами дослідження.

2 ОПИС ПРИЙНЯТИХ ПРОЕКТНИХ РІШЕНЬ

2.1 Аналіз та вибір графової СУБД для дослідження

Графові СУБД – це спеціалізовані системи керування базами даних, оптимізовані для зберігання та обробки даних у вигляді графів. Вони особливо ефективні для роботи з даними, що мають велику кількість зв’язків. До основних графових СУБД, доступних на теперішній час, відноситься:

- Neo4j – одна з найпопулярніших графових СУБД, є високо продуктивною та масштабованою [6];
- Amazon Neptune – графова база даних з відкритим вихідним кодом від Amazon Web Services [7];
- TigerGraph – графова база даних, розроблена для проведення масштабної аналітики в реальному часі [8];
- OrientDB – мультимодельна база даних, підтримує графову модель поряд з документною та об'єктною моделями [9];
- ArangoDB – мультимодельна база даних, поєднує графову, документну та ключ-значення моделі [10].

Для вибору оптимальної графової бази даних для дослідження використовуватимемо наступні критерії:

- популярність СУБД – це важливий фактор, який корелює з розміром спільноті розробників, кількістю доступних ресурсів для навчання та підтримки, довгостроковою перспективою розвитку продукту. Популярність СУБД буде визначатися за допомогою оцінок з порталу db-engines.com;
- підтримка атрибутів вершин та ребер – важливий критерій, оскільки при трансофрмації даних у графову модель постає необхідність зберігання атрибутів як для вершин, так і для ребер графу: повна підтримка атрибутів дозволяє створювати більш інформативні графові моделі;
- підтримка суто графової моделі даних – даний критерій оцінює наскільки СУБД орієнтована саме на роботу з графами. Суто графова

модель забезпечує кращу продуктивність, що особливо важливо при міграції складних структурованих даних;

- підтримка мов запитів для графів – спеціалізовані мови запитів, такі як Cypher або SPARQL, дозволяють ефективно розробляти та виконувати складні запити на графовій моделі даних;
- масштабованість – здатність СУБД ефективно працювати з великими обсягами даних є важливим фактором, оскільки міграція структурованих даних може призвести до створення дуже великих графів;
- простота розгортання – критерій, який впливає на швидкість початку роботи з СУБД та витрати на адміністрування. Враховується складність процесу встановлення, налаштування та підготовки системи до роботи, кількість необхідних конфігурацій та додаткових компонентів.

Наведемо шкали для обраних критеріїв:

а) популярність СУБД: дані з порталу db-engines.com, приймає значення від 0 до ∞ (дані актуальні на жовтень 2024 року);

б) підтримка атрибутів вершин та ребер:
   1) повна підтримка для вершин і ребер – 3 бали;
   2) повна підтримка для вершин, обмежена для ребер – 2 бали;
   3) обмежена підтримка для вершин – 1 бал;

в) підтримка суто графової моделі даних:
   1) чиста графова модель – 3 бали;
   2) переважно графова з додатковими можливостями – 2 бали;
   3) мультимодельна з підтримкою графів – 1 бал;

г) підтримка мов запитів для графів:
   1) підтримка Cypher або SPARQL – 3 бали;
   2) удосконалена версія існуючої мови (наприклад SQL) – 2 бали;
   3) власна мова запитів – 1 бал;

д) масштабованість:
   1) горизонтальна та вертикальна – 3 бали;
   2) тільки вертикальна – 2 бали;

3) обмежена– 1 бал;

е) простота розгортання:

1) просте розгортання (наявність керованого хмарного сервісу, автоматизованої конфігурації за замовченням) – 3 бали;
2) помірна складність (часткова автоматизація розгортання, потребує ручне налаштування базової конфігурації) – 2 бали;
3) складне розгортання (складна архітектура з багатьма компонентами, специфічні вимоги до системного оточення, ручне налаштування більшості параметрів) – 1 бал.

Враховуючи вищезазначені критерії та їх шкали, можна побудувати таблицю графових СУБД, в якій буде наведено значення кожного критерію для певної СУБД (див. табл. 2.1).

Таблиця 2.1 – Графові СУБД із визначеними критеріями та їх значеннями (таблиця виконана самостійно)

	Популярність	Атрибути вершин і ребер	Графова модель	Мова запитів	Масштабованість	Простота розгортання
Neo4j	42.51	3	3	3	3	1
Amazon Neptune	2.17	3	2	3	3	3
Tiger Graph	1.46	3	3	1	3	2
Arango DB	3.44	2	1	2	3	3
OrientDB	3.03	3	1	2	2	2

Деякі характеристики, які є спільними для всіх розглянутих графових СУБД, не були включені до порівняльного аналізу. Ці характеристики являють собою базовий функціонал, який очікується від будь-якої графової СУБД [11]:

- підтримка ACID транзакцій;
- підтримка індексації;
- можливіть резервного копіювання та відновлення;
- можливість інтеграції з іншими системами;
- наявність механізмів забезпечення безпеки;
- можливість візуалізації графів.

Для оцінки та порівняння графових СУБД доречно використати метод лінійної адитивної згортки з ваговими коефіцієнтами та нормуючими множниками. Цей метод було обрано з наступних причин:

- дозволяє працювати з кількісними критеріями, що мають різні одиниці вимірювань;
- дозволяє легко налаштовувати важливість різних критеріїв через вагові коефіцієнти;
- доволі простий в розумінні та інтерпретації результатів.

Проведемо розрахунки за формулою згортки 2.1:

$$Z = max \sum_{j=1}^{n} \alpha_j \beta_j a_{ij} \ \forall \ i = \overline{1, m}, \tag{2.1}$$

$$\alpha_j = \frac{1}{\sum_{i=1}^{m} a_{ij}}$$

де α_j – нормуючі множники,

β_j – вагові коефіцієнти.

Для визначення вагових коефіцієнтів будемо спиратися на специфіку дослідження з міграції реляційних даних у графову модель. Підтримка атрибутів вершин та ребер має таку саму важливість, як і підтримка саме графової моделі, адже це ключові фактори для дослідження і вони в 2.5 рази більш важливі, ніж масштабованість. Популярність СУБД та підтримка мов запитів для графів є також важливими показниками і вони в 1.5 рази більш важливі, ніж

масштабованість. Простота розгортання має таку саму важливість як і масштабованість. За пропорційним методом розрахуємо вагові коефіцієнти:

- популярність СУБД: 0.15;
- підтримка мов запитів: 0.15;
- підтримка атрибутів вершин та ребер: 0.25;
- підтримка суто графової моделі даних: 0.25;
- масштабованість: 0.1;
- простота розгортання: 0.1.

Враховуючи нормуючи множники та вагові коефіцієнти, побудуємо таблицю нормалізованих значень критеріїв із відповідними результатами функції згортки для кожної СУБД (див. табл. 2.2).

Таблиця 2.2 – Результати функції згортки для графових СУБД (таблиця виконана самостійно)

	Популярність	Атрибути вершин і ребер	Графова модель	Мова запитів	Масштабованість	Простота розгортання	Z
Neo4j	1	1	1	1	1	0	0.32
Amazon Neptune	0.017	1	0.5	1	1	1	0.2
TigerGraph	0	1	1	0	1	0.5	0.185
ArangoDB	0.048	0	0	0.5	1	1	0.15
OrientDB	0.038	1	0	0.5	0	0.5	0.145
Вагові коефіцієнти	0.15	0.25	0.25	0.15	0.1	0.1	
Нормуючий множник	0.019	0.071	0.1	0.09	0.071	0.09	

На основі проведеного аналізу Neo4j отримала найвищу оцінку (0.32) серед розглянутих графових СУБД, адже вона має наступні переваги:

– є найпопулярнішою графовою СУБД;
– забезпечує повну підтримку атрибутів для вершин і ребер;
– забезпечує підтримку суто графової моделі;
– забезпечує підтримку мови Cypher;
– має високу масштабованість.

Отже, Neo4j є оптимальним вибором для дослідження міграції даних з реляційної у графову БД.

2.2 Аналіз та вибір структур вхідних та вихідних даних

Спочатку розглянемо структуру вихідних даних – графову модель бази даних Neo4J.

Структура даних Neo4j базується на графовій моделі та складається з вузлів та ребер. Вузли використовуються для опису сутностей домену, вони можуть мати властивості (атрибути), унікальний ідентифікатор. Ребра використовуються для встановлення зв’язків між вузлами, мають направлений характер, можуть мати власні атрибути та завжди містять тип, який описує характер зв'язку.

Властивості зберігаються як пари «ключ-значення» і можуть бути присутніми як на вузлах, так і на ребрах. Вони підтримують різні типи даних, включаючи числа, рядки, булеві значення та масиви.

Така архітектура забезпечує гнучкість та ефективність зберігання складних взаємопов'язаних даних, що робить Neo4j потужним інструментом для роботи з графовими базами даних [12].

Вибір об'єктів дослідження для перетворення в графові структури ґрунтувався на аналізі сучасних трендів у сфері зберігання даних. Було прийнято рішення зосередитися на реляційних базах даних як найбільш репрезентативному формату, що використовується в реальних проектах.

Реляційні бази даних були обрані через їх популярність та широке розповсюдження, адже приблизно 60% програмних систем все ще використовують реляційні СУБД [13].

Дані в реляційних БД мають чітку структуру з визначеними схемами, що дозволяє розробити більш точні алгоритми міграції. Також реляційні СУБД підтримують різні види зв’язки між сутностями (один-до-одного, один-до-багатьох), що робить їх цікавим випадком для перетворення в графову модель. Також реляційні БД підтримують SQL, що буде корисним під час написання запитів для перевірки коректності міграції даних.

В якості реляційної бази даних для проведення дослідження було обрано Microsoft SQL Server.

Дана СУБД має наступні переваги [14]:

- одна з найпопулярніших реляційних СУБД в світі;
- висока стабільність роботи та відмовостійкість;
- висока продуктивність навіть при значних обсягах даних;
- має розвинену інтеграцію з екосистемою .NET, що дозволяє ефективно працювати з БД на мові програмування C#;
- добре масштабується для роботи з великими обсягами даних;
- підтримка резервного копіювання та відновлення даних;
- має обширну та деталізовану документацію та велику спільноту розробників.

Таким чином, вибір реляційних баз даних як структур вхідних даних для дослідження методів міграції в графову модель є обґрунтованим та дозволяє провести всебічний аналіз проблеми.

Реляційна база даних Microsoft SQL Server має структуру, засновану на реляційній моделі даних. Основними елементами є таблиці, які складаються з рядків та стовпців. Кожна таблиця має чітко визначену схему зі стовпцями певних типів даних. Первинні ключі унікально ідентифікують кожен рядок у таблиці, а зовнішні ключі встановлюють зв'язки між різними таблицями.

2.3 Розробка математичної моделі міграції даних

Розглянемо математичну постановку задачі дослідження.

Нехай задано організаційну частину моделі даних D, яку необхідно мігрувати у графову структуру G. Організаційну частину D реляційної моделі БД можна представити у вигляді кортежу за формулою 2.2:

$$D =< T, P, R >, \tag{2.2}$$

де $T = \{T_i | i = \overline{1,n}\}$ – скінченна множина таблиць реляційної БД;

$P = \{P_i | i = \overline{1,n}\}$ – скінченна множина атрибутів в БД, де $P_i = \{P_{ij} | j = \overline{1,m}\}$ – скінчення множина атрибутів таблиці T_i;

$R \subseteq \mathrm{T} \times \mathrm{T}$ – множина зв'язків між таблицями.

Граф G, що буде результатом міграції, представлено у вигляді формули 2.3:

$$G =< V, E >, \tag{2.3}$$

де V – множина вершин графа, $V = \{V_l(PV_l) | l = \overline{1,p}\}$, $V_l(PV_l)$– скінченна множина вершин певного l-ого типу, яка характеризується однаковою множиною атрибутів PV_l;

$E \subseteq \mathrm{V} \times \mathrm{V}$ – множина ребер графа, $E = \{E_k(PE_k) | k = \overline{1,q}\}$, $E_k(PE_k)$ – скінченна множина ребер певного k-ого типу, яка характеризується однаковою множиною атрибутів PE_k ребер.

Процес міграції можна представити як сукупність відображень:

– $\varphi: T \to V \cup E$ – відображення, що ставить у відповідність кожній таблиці $T_i \in T$ множину вершин $V_i(PV_i) \in V$ певного типу або множину ребер $E_i(PE_i) \in E$ певного типу;

– $\Psi: \mathrm{P} \to PV \cup PE$ – відображення, що ставить у відповідність кожній множені атрибутів $P_i \in P$ таблиці T_i множину PV_i атрибутів вершин або множину PE_i атрибутів ребер;
– $\Omega: \mathrm{R} \to E$ – відображення, що ставить у відповідність кожному зв'язку $(T_i, T_j) \in R$ множину ребер $(V_l, V_k) \in E$ одного типу.

Отже, для власного методу міграції реляційної БД в графову БД під Neo4J необхідно визначити відображення φ, Ψ та Ω, що задають правила перетворення таблиць, зв'язків та атрибутів реляційної БД в множини вершин та ребер певних типів графової БД, та їх атрибути.

2.4 Аналіз та вибір існуючих методів міграції структурованих даних

Наведемо опис знайдених методів конвертації структурованих даних в графову модель.

Розглянемо метод міграції Rel2Graph [15].

Першим кроком метод визначає, які таблиці є таблицями сутностей, а які таблиці використовуються для створення зв'язків (наприклад, проміжна таблиця в зв'язку багато до багатьох).

Таблиця кваліфікується як таблиця сутностей, якщо вона задовольняє будь-якій з наступних трьох умов:

– таблиця позбавлена будь-якого зовнішнього ключа;
– таблиця або має один зовнішній ключ, або містить більше двох зовнішніх ключів, тобто кількість зовнішніх ключів не дорівнює двом;
– таблиця має два зовнішні ключі та первинний ключ, що складається лише з одного стовпця.

Згодом ці таблиці сутностей транслюються у вузли графа, де кожен вузол позначається відповідною назвою таблиці сутностей. Кожен рядок у таблиці сутностей відповідає вузлу, тоді як стовпці в таблицях сутностей перетворюються на властивості (атрибути) вузлів.

Таблиця стає таблицею зв'язування, якщо кількість зовнішніх ключів дорівнює двом і виконується одна з двох додаткових умов:

– кількість первинних ключів не дорівнює одиниці;
– первинний ключ складається з двох зовнішніх ключів.

Таблиці зв’язування стають ребрами графа. Кожне ім’я таблиці представлено типом ребра. Кожен рядок у таблиці зв’язування є ребром. Стовпці зв’язувальних таблиць стають властивостями (атрибутами) ребер.

Після того, як визначені таблиці сутностей та таблиці зв’язування, необхідно пройтися по зовнішнім ключам таблиць та проставити додаткові ребра графа з типом *_HAS_*, якщо вони ще не існують (міграція зв’язку один до одного або один до багатьох).

Даний метод має певні недоліки:

– таблиці, що мають два зовнішніх ключа та простий первинний ключ рахуються як таблиці сутності, хоча дуже часто зустрічаються таблиці зв’язування з подібною структурою, які мають лише три стовбці – первинний ключ (частіше за все генерується автоматично) та два зовнішніх ключа (посилання на дві таблиці, зв’язок багато до багатьох);
– таблиці, що мають два зовнішніх ключа та складовий первинний ключ рахуються як таблиці зв’язування, хоча можливі таблиці сутностей, які мають два зовнішніх ключа та складовий первинний ключ;
– метод приймає на вхід файл з SQL-скриптами для створення реляційної бази даних, а не підключається до існуючої бази даних, витрачаючи таким чином час на обробку та парсінг текстового файлу.

Перейдемо до розгляду наступного методу міграції [16].

Даний метод можна поділити на дві частини. В першій частині із кожної таблиці зчитується інформація про первинні та зовнішні ключі, а також зберігаються зв’язки у вигляді списку батько-дитина. Якщо тип сутності має зовнішній ключ, він має назву дочірній, а тип сутності, на який посилається, називається батьківським.

Після цього для кожної сутності визначаються правила конвертації в графову модель. Для цього перевіряються наступні умови:

а) якщо у сутності відсутні зовнішні ключі та немає жодних зв’язків батько-дитина (тобто на цю таблицю жодна інша таблиця не посилається), то така таблиця стає вузлом графу;

б) якщо у сутності відсутні зовнішні ключі, але є зв’язки батько-дитина, виконуються наступні дії:

1) пошук дочірньої сутності;
2) якщо всі первинні ключі дочірньої сутності є зовнішніми ключами, то така сутність вважається сутністю зв’язування (проміжна таблиця для зв’язку багато-до-багатьох) та така сутність перетворюється на ребра графу;
3) інакше між дочірньою та батьківською сутністю встановлюється зв’язок один до одного/багатьох;

в) якщо у сутності є зовнішні ключі та є зв’язки батько-дитина, то:

1) якщо всі первинні ключі сутності є зовнішніми ключами, то така сутність вважається сутністю зв’язування та така сутність перетворюється на ребра графу;
2) інакше відбувається пошук батьківської сутності по відношенню до поточної сутності та між ними встановлюється зв’язок один до одного/багатьох.

Основними недоліками такого підходу є:

– сутністю зв’язування вважається та сутність, у якої всі первинні ключі є зовнішніми ключами, що не завжди є вірним;
– наявність додаткового списку, який зберігає відношення батько-дитина;
– метод приймає на вхід файл з ER-діаграмою, що потребує додаткового часу на її обробку.

Враховуючи недоліки вищезазначених методів пропонується створення удосконаленого методу міграції, який би мав наступні переваги:

– робота з реляційною базою даних напряму, без використання та обробки додаткових файлів;

- врахування різних випадків для таблиця зв’язування: з наявним простим первинним ключом та без нього;
- врахування таблиць з двома зовнішніми ключами та складовим первинним ключом: таблиця буде вважатися таблицею зв’язування, якщо загальна кількість стовбців в такій таблиці менше ніж 4, тому що частіше за все таблиці зв’язування мають невелику кількість стовбців (наприклад, 2 стовбці з зовнішніми ключами, які є складовим первинним ключом, і ще максимум 1-2 стовбці для додаткової інформації);
- підвищення швидкодії алгоритму шляхом використання паралелізму, наприклад, під час заповнення даними графової СУБД;
- можливість задання певних правил та обмежень користувачем, які будуть враховані під час здійснення конвертації, наприклад, задання таблиць, які повинні рахуватися як таблиці зв’язування.

Отже, було проаналізовано існуючі методи міграції, виявлено їх недоліки та запропоновано рішення для їх удосконалення.

2.5 Опис розробленого методу міграції

Запропонований удосконалений метод міграції структурованих даних має на меті подолати недоліки попередніх методів та забезпечити більш гнучкий та ефективний процес конвертації (див. рис. 2.1).

Розглянемо його алгоритм, який складається з наступних кроків.

На першому етапі виконується аналіз структури реляційної бази даних. Для цього здійснюється підключення безпосередньо до бази даних Microsoft SQL Server. Для встановлення з’єднання з СУБД використовується рядок з’єднання (connection string).

Під час аналізу система збирає детальну інформацію про кожну таблицю, включаючи:

- назви таблиці та стовпців;
- тип даних стовпців;
- первинні ключі;

– зовнішні ключі.

Зібрана інформація зберігається у таких структурах даних як списки, хеш-сети та словники.

На кроці 2 відбувається врахування налаштувань користувача, який може задати особливості предметної області.

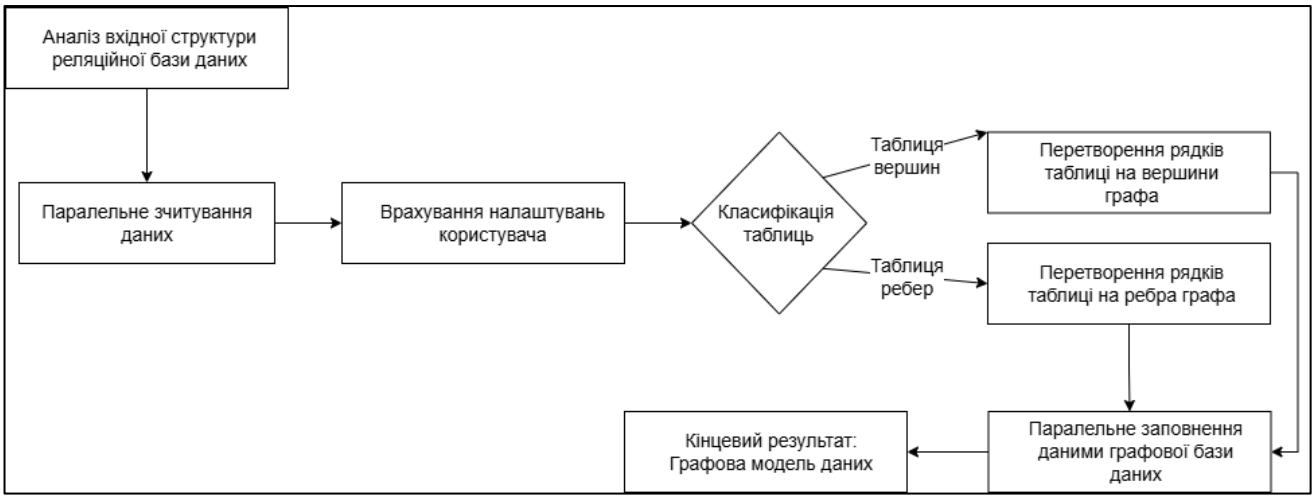


Рисунок 2.1 – Графічне представлення удосконаленого алгоритму (рисунок виконаний самостійно)

Система дозволяє:

– визначати правила перетворення для специфічних таблиць;

– виключати певні таблиці або стовпці з процесу конвертації.

На кроці 3 відбувається класифікація таблиць за їх призначенням та структурою.

Перш ніж описувати правила класифікації, наведемо структуру типових таблиць, що реалізують зв’язок багато-до-багатьох:

– таблиця має простий первинний ключ та два зовнішніх ключа, які посилаються на відповідні таблиці. Також інколи може бути присутній деякий змістовний атрибут;

– таблиця має два зовнішніх ключа, які одночасно є складовим первинним ключом. Такі таблиці також можуть містити додатковий змістовний атрибут.

Відповідно, якщо кількість змістовних атрибутів у таких таблицях більше одного, тоді такі таблиці будуть вважатися таблицями сутностей.

Тепер наведемо безпосередньо правила класифікації таблиць.

Таблиця вважається таблицею сутності, якщо:

- зовнішні ключі відсутні;
- кількість зовнішніх ключів дорівнює одному, або більше ніж два;
- існує два зовнішніх ключа та простий первинний ключ, причому кількість стовпців в таблиці більше ніж чотири, тобто в наявності є мінімум два змістовних атрибути;
- існує два зовнішніх ключа та складовий первинний ключ, причому кількість стовпців в таблиці більше ніж три, тобто в наявності є мінімум два змістовних атрибути.

Таблиця вважається таблицею зв’язування, якщо:

- існує два зовнішніх ключа та простий первинний ключ, причому кількість стовбців не більше ніж чотири, тобто в наявності є лише один змістовний атрибут;
- існує два зовнішніх ключа та складовий первинний ключ, причому кількість стовбців не більше ніж три, тобто зовнішні ключі є складовим первинним ключом та в наявності може бути лише один змістовний атрибут.

Крок 4 пов’язаний з перетворенням рядків таблиць вершин у вершини графової БД.

На цьому кроці відбувається перетворення таблиць сутностей у вузли графової бази даних. Кожна таблиця сутностей стає типом вузла, а кожен її рядок перетворюється на окремий вузол.

Властивості вузлів формуються на основі стовпців вхідної таблиці. Система зберігає всю семантичну інформацію, включаючи числові, текстові та інші типи даних.

Крок 5 пов’язаний з перетворенням рядків таблиць ребер у ребра графа.

Таблиці зв'язування перетворюються на ребра графа. Назва таблиці зв'язування стає типом ребра, а її рядки – безпосередньо ребрами між вузлами. Якщо в таблиці зв’язування були присутні змістовні атрибути, то вони перетворюються на атрибути ребер.

На етапі перетворення рядків таблиць ребер у ребра графа також створюються ребра типу HAS для всіх зовнішніх ключів, що знаходяться у таблицях вершин. Кожен зовнішній ключ стає ребром, яке з'єднує поточну вершину (рядок таблиці із зовнішнім ключем) з відповідною цільовою вершиною (рядок таблиці, на який посилається зовнішній ключ).

Після цього необхідно пройтися по зовнішнім ключам, які не входять до таблиць зв’язування, та на їх основі створити ребра між відповідними вузлами.

Крок 6 реалізує паралелізацію перетворення, яка є ключовим елементом підвищення продуктивності алгоритму.

Для реалізації паралелізації використовуються наступні підходи.

Паралельне зчитування даних – система розподіляє таблиці між декількома потоками, дозволяючи одночасно обробляти різні таблиці. Кожен потік незалежно виконує аналіз структури та її класифікацію.

Паралельне заповнення графової бази даних – створення вузлів та ребер відбувається паралельно. Це досягається шляхом розподілу даних на незалежні групи, які можуть оброблятися одночасно.

Отже, запропонований удосконалений алгоритм конвертації являє собою потужний та гнучкий інструмент для перетворення реляційних даних у графову модель.

2.6 Аналіз та моделювання предметної області

В якості предметної області для дослідження було обрано соціальну мережу. Цей вибір зумовлений наступними факторами:

– соціальні мережі є одними з найпопулярніших інтернет-сервісів сьогодні, якими користуються мільйони людей щодня. Дослідження методів ефективного зберігання та обробки даних соціальних мереж має

високу практичну цінність для розробки сучасних масштабованих веб-застосунків;

– соціальні мережі складаються з великої кількості взаємопов’язаних сутностей та їх атрибутів, тому така предметна область підходить для моделювання в графових базах даних.

Для представлення концептуальної моделі предметної області соціальної мережі було розроблено наступну ER-діаграму (див. рис. 2.2) [17].

Ця модель складається з наступних ключових сутностей:

– користувач – містить системні дані про користувачів програмної системи;
– профіль користувача – містить додаткову персональну інформацію про користувачів, включаючи навики та хобі;
– дружба – двосторонні зв’язки дружби між користувачами;
– пости – елементи, що створюються користувачами соціальної мережи;
– коментарі – реакції користувачів у вигляді текстових повідомлень;
– лайки – уподобання користувачів;
– хештеги – тематичні мітки посту;
– чати – приватні діалоги між коритсувачами;
– повідомлення – основний елемент чату;
– чорний список – список заблокованих користувачів.

Наведена ER-діаграма відображає ключові аспекти соціальної мережі та може бути трансформована в логічну модель даних для подальшого впровадження.

На основі розробленої ER-діаграми розробимо логічні моделі для реляційної та графової баз даних.

Під час перетворення ER-діаграми в реляційну модель було виконано наступні дії [18]:

– сутності ER-діаграми було перетворено в таблиці реляційної бази даних;

– для моделювання зв’язків багато-до-багатьох було впроваджено проміжні таблиці;

– для моделювання зв’язків один-до-багатьох було впроваджено додаткові навігаційні атрибути (зовнішні ключі).

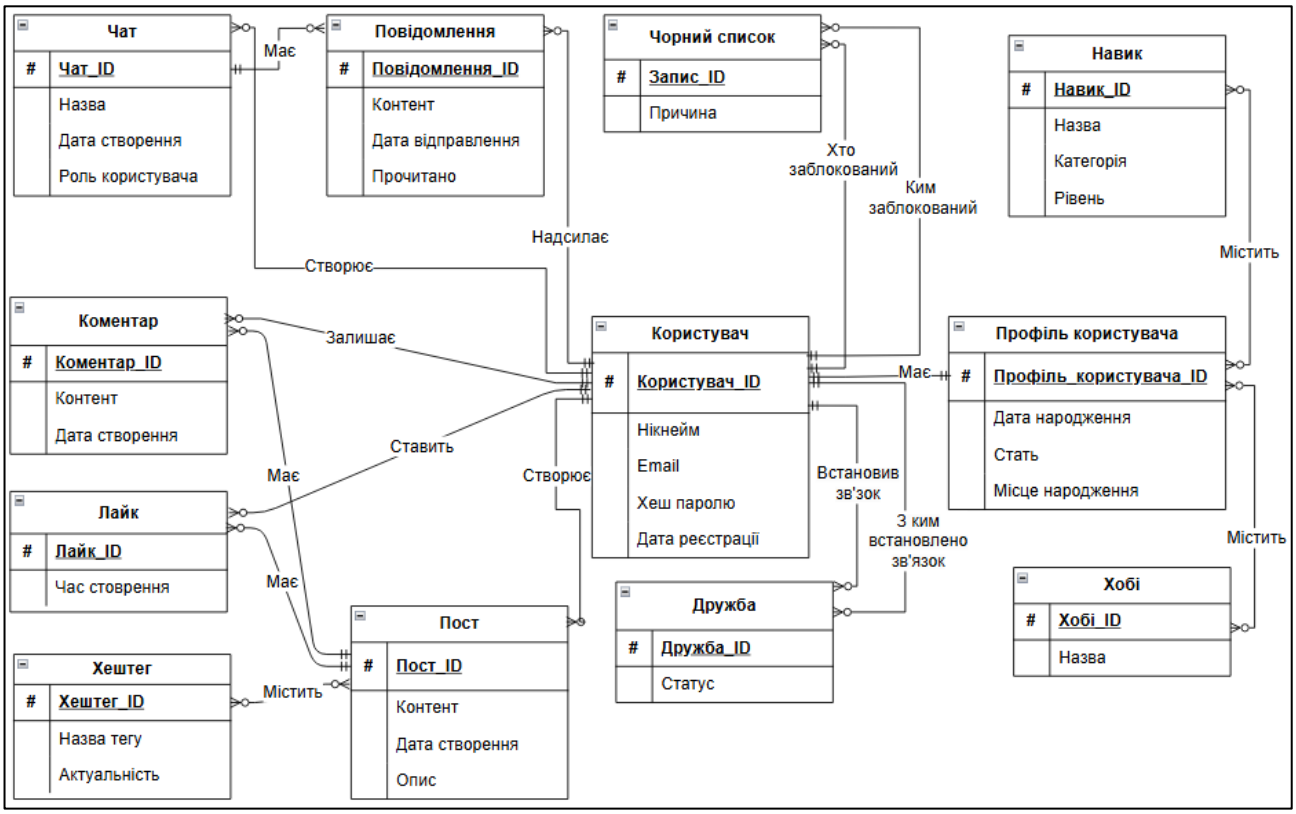


Рисунок 2.2 – ER-діаграма бази даних предметної області (рисунок виконаний самостійно)

В результаті було отримано наступну реляційну логічну модель бази даних (див. рис. 2.3).

Перейдемо до проєктування графової логічної моделі.

Слід зазначити, що на сьогоднішній день не існує стандартизованої нотації до побудови логічної моделі графових баз даних, тому було вирішено використовувати модифікацію реляційної.

Для відображення атрибутів ребер було використано прозорі прямокутники.

Проміжні сутності, які використовувалися для створення зв’язку багато-до-багатьох, було перетворено на ребра графу.

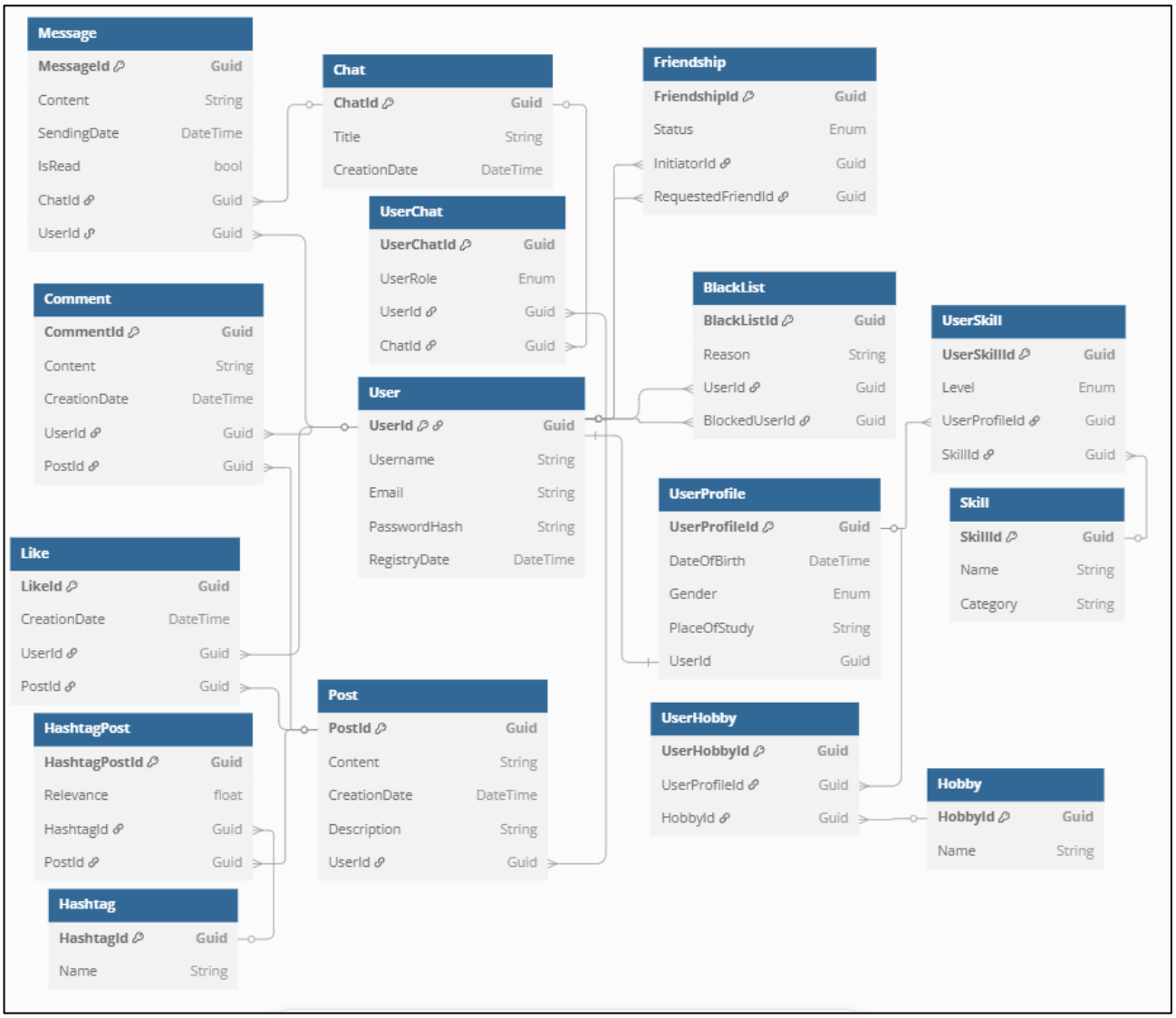


Рисунок 2.3 – Схема реляційної бази даних предметної області (рисунок виконаний самостійно)

На рисунку 2.4 наведено спроектовану логічну модель графової БД.

Також в якості предметної області можуть бути розглянуті ігрові серверні системи [19]. В якості прикладу розглянемо гру жанру action-adventure з елементами RPG та виділеним сервером.

Для такої ігри база даних повинна зберігати інформацію про обліковий запис гравця, стан та інформацію його персонажів в ігровому світі, список завдань, противників та неігрових персонажів, історію подій у грі.

ER-діаграму, а також логічну графову модель для предметної області ігрової серверної системи [19] наведено в додатку Е.

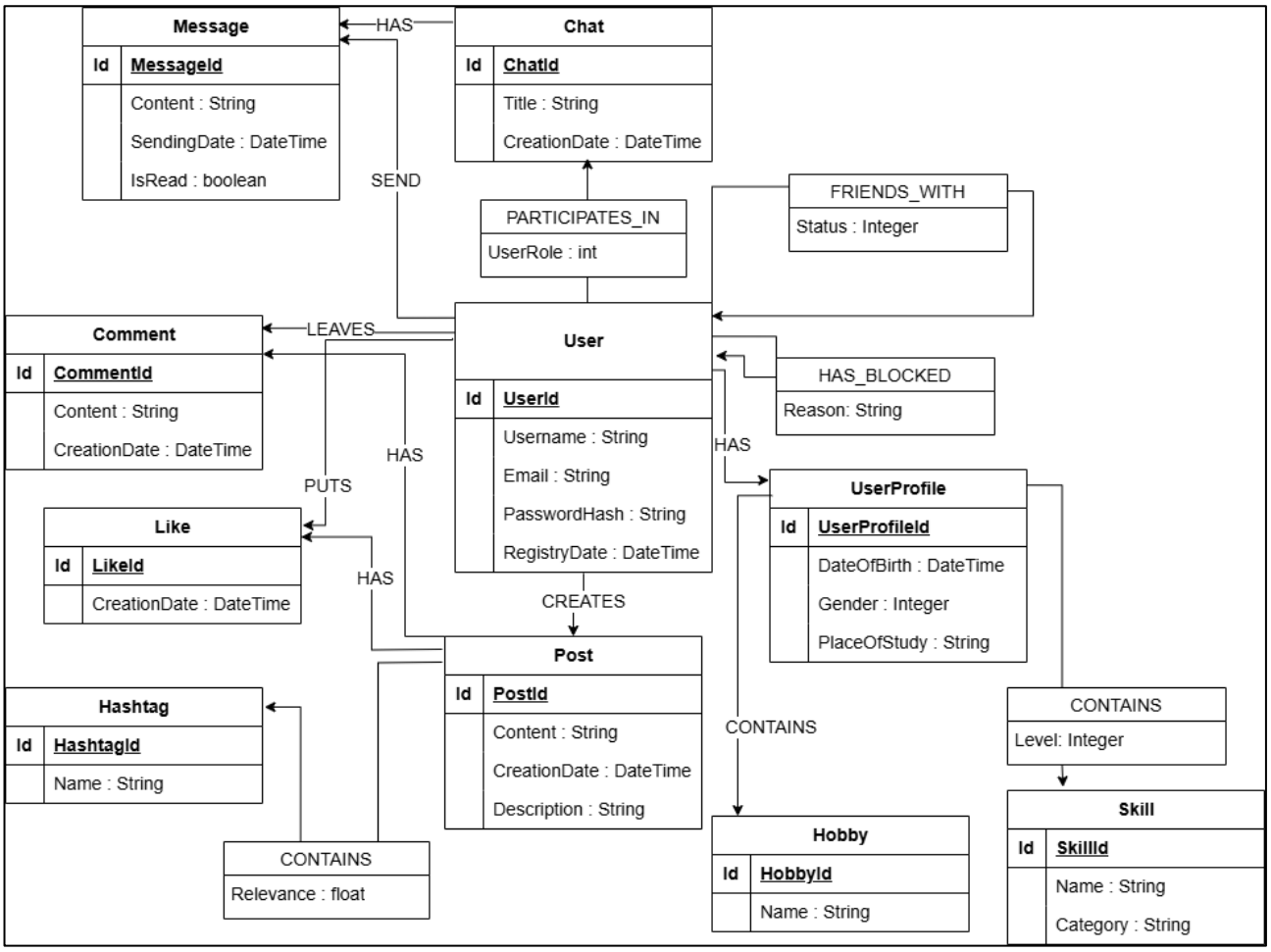


Рисунок 2.4 – Логічна модель графової бази даних предметної області (рисунок виконаний самостійно)

Для повноти експериментів було розглянуто наступні предметні області, для яких було розроблено відповідні БД:

- Ecommerce для предметної області електронної комерції (9 таблиць, 9500098 рядків);
- SocialNetwork для предметної області соціальних мереж (16 таблиць, 6744 рядків);
- GameDB для предметної області ігрових серверних систем (21 таблиця, 7885 рядків);
- TravelingDb для предметної області проведення подорожей (4 таблиці, 136 рядків);

- ReportDb для предметної області волонтерської діяльності (4 таблиці, 45 рядків);
- SportDb для предметної області спортивних змагань (4 таблиці, 70 рядків).

Спроектовані логічні моделі будуть використанні для створення фізичних моделей БД.

2.7 Планування експериментального дослідження

Для планування експериментального дослідження необхідно визначити метрики для оцінки ефективності алгоритмів міграції даних з реляційної у графову БД та якості отриманих моделей.

Для оцінки ефективності алгоритмів будуть використані наступні метрики:

- час міграції даних – час, який витрачає розроблений алгоритм на перетворення структурованих даних у графову модель. Час буде замірятися для різних обсягів вхідних даних, що дозволить проаналізувати масштабованість розроблених алгоритмів;
- обсяг використаної оперативної пам’яті – кількісна оцінка обсягу пам’яті, що споживає алгоритм під час процесу міграції. Ця метрика дозволить порівняти ресурсоємність розроблених алгоритмів.

Для оцінки якості та повноти отриманих графових моделей буде використана метрика семантичної еквівалентності запитів. Сутність цієї метрики полягає у порівнянні результатів ідентичних за логікою запитів, сформованих для вхідної та вихідної (графової) структури даних.

Наведена методологія для оцінки якості результуючих графових моделей включає:

- формування набору тестових запитів для вхідних структур;
- трансформацію цих запитів на мову Cypher для графової бази даних Neo4j;
- порівняння результатів тестових запитів за наступними критеріями: кількість результуючих записів, ідентичність значень атрибутів.

Кількісна оцінка (коефіцієнт семантичної відповідності) може розраховуватися за формулою 2.4:

$$K_{\text{сем}} = \frac{N_{\text{вихідні}}}{N_{\text{вхідні}}} \tag{2.4}$$

де $N_{\text{вихідні}}$ – кількість ідентичних записів у вихідній структурі,

$N_{\text{вхідні}}$ – кількість ідентичних записів у вхідній структурі.

Результуюча графова модель є якісною, якщо коефіцієнт семантичної відповідності дорівнює одиниці.

Також є сенс проводити замір часу виконання тестових запитів, щоб оцінити доцільність конвертації вхідних структур у графову.

Для порівняльного аналізу ефективності запропонованого методу міграції та існуючих підходів планується оцінка продуктивності роботи з отриманими графовими моделями. Оцінка буде проводитися шляхом виконання еквівалентних запитів на графових структурах, створених різними алгоритмами міграції. Основні метрики продуктивності для порівняння включають:

- час виконання запитів;
- завантаженість системних ресурсів (завантаженість процесора під час виконання запитів, обсяг споживання оперативної пам'яті);
- ефективність зберігання даних (загальний обсяг бази даних на диску).

Не менш важливою метрикою якості міграції даних є результуюча кількість вузлів та зв’язків в графовій БД.

2.8 Розробка запитів для експериментального дослідження

Для перевірки коректності міграції даних з реляційної бази даних в графову необхідно розробити запити на мовах SQL та Cypher. Результати відповідних запитів будуть порівнюватися між собою та в залежності від результатів порівняння буде зроблено висновок про повноту міграції даних. Таким чином

алгоритми будуть порівнюватися за метрикою семантичної еквівалентності запитів, опис якої наведено в розділі 2.7.

Для коректного аналізу результатів міграції розроблені запити повинні відповідати наступним вимогам:

- повинні повертати семантично ідентичні результати для обох баз даних;
- кількість рядків та значень у результатах повинна збігатися;
- за наявності агрегації, розрахункові значення повинні бути ідентичними;
- повинні охоплювати різні типи зв'язків, присутні в схемі даних;
- мають тестувати таблиці/вузли з різною кількістю атрибутів;
- повинні відображати типові сценарії використання для конкретної предметної області;
- мають бути придатними для виконання на різних обсягах даних;
- мають перевіряти відмінності в підходах до міграції.

Враховуючи вищезазначені вимоги, було розроблено наступні запити для предметної області ігрових серверних додатків:

- отримання інформації про екіпіровку персонажа;
- аналіз предметів інвентаря персонажів;
- комплексний аналіз квестів, перегляд інформації про нагороди за квести;
- отримання інформації про лут персонажа;
- інші тестові запити.

Для предметної області соціальних мереж було розроблено запити:

- отримання інформації про пости користувачів;
- отримання інформації про пости з актуальними хештегами;
- аналіз навичок користувачів;
- пошук спільних друзів для певного користувача;
- інші тестові запити.

Такий набір запитів дозволить провести ґрунтовний аналіз обох алгоритмів міграції та допоможе визначити, який підхід є більш ефективним для різних сценаріїв використання.

2.9 Проєктування програмного забезпечення для експерименту

Для проведення експериментів необхідно розробити програмний додаток, який би виконував міграцію даних з реляційної БД в графову різними алгоритмами та збирав при цьому певні метрики. В якості технологій для реалізації додатку було обрано платформу .NET 8 та мову програмування C#, адже вони мають низку переваг:

- наявність великої кількості бібліотек, в тому числі для роботи з реляційними та графовими БД;
- кросплатформеність, що дозволяє за необхідністю проводити експерименти на різних операційних системах;
- наявність потужних інструментів діагностики та аналізу, таких як profilers та debugging tools;
- висока продуктиівність завдяки CLR.

В якості архітектури програмного забезпечення було обрано модифіковану тришарову архітектуру (див. рис. 2.5).

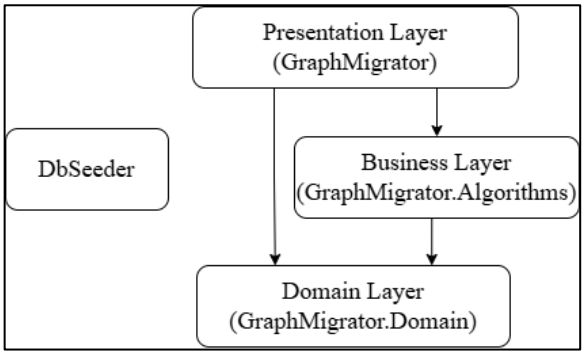


Рисунок 2.5 – Архітектура програмного забезпечення (рисунок виконаний самостійно)

Архітектура складається з наступних складових:

- шар представлення – відображає результати експериментів в консольному вікні;
- шар бізнес-логіки – містить в собі реалізації всіх необхідних алгоритмів для міграції даних;

– шар домену (ядра) – містить сутності, моделі та конфігурацію додатку;
– додаток для наповнення реляціних баз даних тестовими даними (DbSeeder) – окрема складова додатку, яка використовується безпосередньо для підготовки експериментів.

Модифікація тришарової архітектури полягає у відсутності окремо виділеного шару доступу до даних. Це обумовлено специфікою досліджуваної предметної області: додаток повинен мігрувати будь-яку реляційну базу даних MSSQL, а отже створення шару доступу до даних для конкретної предметної області неможливо. Замість цього було створено шар домену, який містить загальні сутності для всіх реляційних баз даних, такі як ColumnSchema, TableSchema та інші. Шар домену для своєї роботи можуть використовувати як шар бізнес-логіки так і шар представлення – це є елементом чистої (onion) архітектури.

Для опису процесу проведення експерименту було створено діаграму діяльності (див. рис. 2.6).

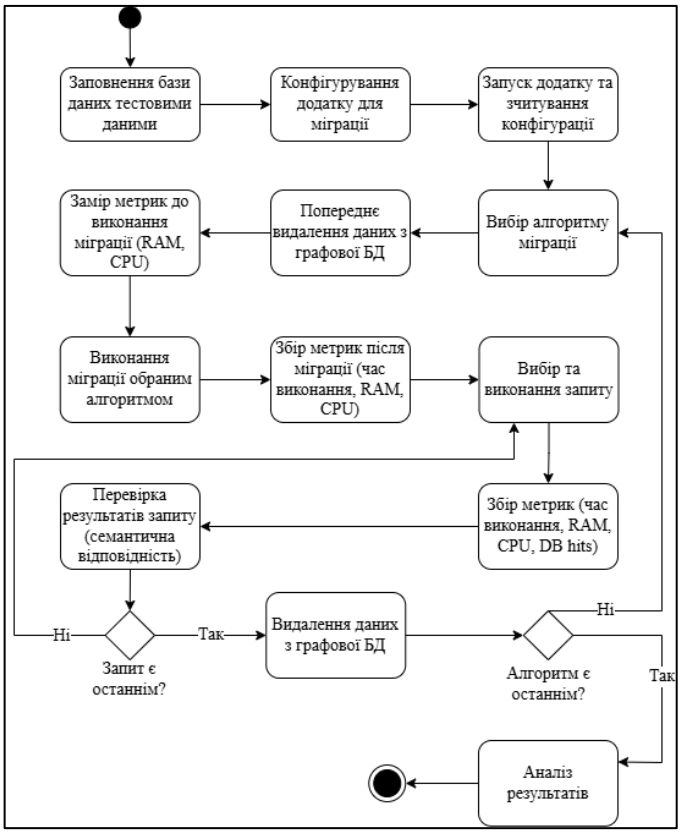


Рисунок 2.6 – Діаграма діяльності проведення експерименту (рисунок виконаний самостійно)

Отже, окрім самого додатку для міграції даних пропонується створити окрему утиліту для заповнення реляційних баз даних тестовими даними, що відповідають обраним предметним областям. Такий інструмент повинен підтримувати такі параметри налаштування:

- рядок з’єднання до цільової БД;
- кількість записів для генерації (для основних таблиць).

Для генерації даних доцільно використовувати бібліотеку Bogus, яка дозволяє задавати правила для генерації реалістичних даних.

Для перевірки результатів запитів також пропонується створити окремий застосунок, який буде приймати SQL та Cypher запити та порівнювати між собою отримані результати.

Додаток для перевірки результатів запитів повинен підтримувати конфігурацію таких параметрів:

- рядок з’єднання до вхідної БД MSSQL;
- параметри підключення до вихідної БД Neo4j;
- кількість знаків після коми для порівняння (для чисел з плаваючою комою).

Для отримання даних та структури вхідної реляційної бази даних MSSQL необхідно використовувати бібліотеку Microsoft.Data.SqlClient, що є удосконаленою версією бібліотеки System.Data.SqlClient.

Для роботи з Neo4J під час міграції даних необхідно використовувати офіційний драйвер Neo4J для .NET – Neo4j.Driver.

Таким чином, в результаті проєктування програмного забезпечення для експерименту було описано та спроектовано три додатки: для заповнення реляційних баз даних тестовими даними, для міграції даних та для перевірки результатів еквівалентних запитів.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Розробка додатку для наповнення реляційних баз даних

Для наповнення вхідних реляційних баз даних тестовими даними було вирішено створити окремий консольний додаток. В якості технологій для реалізації такого додатку було обрано платформу .NET 8 та мову програмування C#.

Створене програмне забезпечення дозволяє згенерувати дані для таких предметних областей:

– соціальна мережа;

– ігровий серверний застосунок.

Перш ніж генерувати дані, необхідно створити тестові бази даних. Для цього було розроблено відповідні SQL скрипти (див. рис. 4.1).

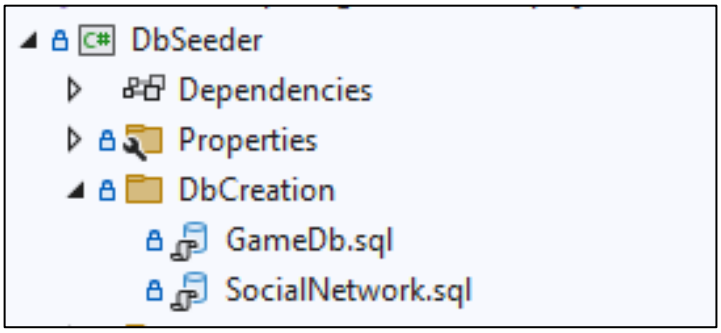


Рисунок 3.1 – Розроблені SQL скрипти для створення тестових БД (рисунок виконаний самостійно)

Після цього було розроблено класи-моделі на мові C#, які відповідають сутностям бази даних. Нижче наведено код такої моделі для сутності пост з предметної області соціальної мережі.

```
public class PostData
{
    public Guid PostId { get; set; }
    public string Content { get; set; }
    public string Description { get; set; }
    public DateTime CreationDate { get; set; }
    public Guid UserId { get; set; }
}
```

Розроблений додаток підтримує конфігурацію наступних параметрів:

– для предметної області соціальної мережі: кількість згенерованих записів для користувачів, постів, коментарів, лайків, хештегів, навичок, хобі, чатів, повідомлень, друзів;
– для предметної області ігрового серверного застосунку: кількість згенерованих записів для користувачів, персонажів, атрибутів, елементів, квестів, противників, неігрових персонажів;
– для обох предметних областей: рядок підключення до цільової БД.

Для генерації даних було використано бібліотеку Bogus. На рисунку 3.2 наведено код для генерації коментарів.

```
1 reference
private async Task SeedComments(SqlConnection connection)
{
    Console.WriteLine($"Seeding {_options.CommentCount} comments...");

    var faker = new Faker<CommentData>()
        .RuleFor(c => c.CommentId, f => Guid.NewGuid())
        .RuleFor(c => c.Content, f => f.Lorem.Sentence())
        .RuleFor(c => c.CreationDate, f => f.Date.Recent(30))
        .RuleFor(c => c.UserId, f => f.PickRandom(_ids["Users"]))
        .RuleFor(c => c.PostId, f => f.PickRandom(_ids["Posts"]));

    var comments = faker.Generate(_options.CommentCount);

    using var transaction = connection.BeginTransaction();
    foreach (var comment in comments)
    {
        _ids["Comments"].Add(comment.CommentId);

        string sql = @"INSERT INTO [Comment] ([CommentId], [Content], [CreationDate], [UserId], [PostId])
                       VALUES (@CommentId, @Content, @CreationDate, @UserId, @PostId)";

        using var command = new SqlCommand(sql, connection, transaction);
        command.Parameters.AddWithValue("@CommentId", comment.CommentId);
        command.Parameters.AddWithValue("@Content", comment.Content);
        command.Parameters.AddWithValue("@CreationDate", comment.CreationDate);
        command.Parameters.AddWithValue("@UserId", comment.UserId);
        command.Parameters.AddWithValue("@PostId", comment.PostId);
        await command.ExecuteNonQueryAsync();
    }

    transaction.Commit();
}
```

Рисунок 3.2 – Генерація коментарів (рисунок виконаний самостійно)

Згенеровані дані повинні відповідати реальним даним, тому для генерації певних сутностей застосовувалася більш складна логіка. Наприклад, під час генерації спорядження для персонажів враховується тип елемента (item), тип слота, рівень персонажа (див. рис. 3.3).

```
private void SeedEquipment()
{
    Console.WriteLine("Seeding character equipment...");

    var slots = new[] { "Head", "Chest", "Legs", "Feet", "Hands", "MainHand", "OffHand", "Ring1", "Ring2", "Necklace" };
    var equipmentFaker = new Faker<EquipmentData>();
    var equipments = new List<EquipmentData>();
    var equipmentId = 1;

    var weaponArmorItems = ExecuteReader<ItemData>("SELECT Id, Type, RequiredLevel FROM Item WHERE Type IN ('Weapon', 'Armor', 'Jewelry')");

    foreach (var characterId in _entityIds["Character"])
    {
        var characterLevel = ExecuteScalar<int>($"SELECT [Level] FROM [Character] WHERE Id = {characterId}");
        var eligibleItems = weaponArmorItems.Where(i => i.RequiredLevel <= characterLevel).ToList();
        if (eligibleItems.Any())
        {
            var slotsToEquip = slots.OrderBy(x => Guid.NewGuid()).Take(_random.Next(3, 9)).ToList();

            foreach (var slot in slotsToEquip)
            {
                string itemType = "Armor";
                if (slot == "MainHand" || slot == "OffHand")
                    itemType = "Weapon";
                else if (slot == "Ring1" || slot == "Ring2" || slot == "Necklace")
                    itemType = "Jewelry";

                var filteredItems = eligibleItems.Where(i => i.Type == itemType).ToList();
                if (filteredItems.Any())
                {
                    var randomItem = filteredItems[_random.Next(filteredItems.Count)];
                    equipments.Add(new EquipmentData
                    {
                        Id = equipmentId++,
                        CharacterId = characterId,
                        ItemId = randomItem.Id,
                        Slot = slot
                    });
                }
            }
        }
    }

    BulkInsert("Equipment", equipments, (equipment, cmd) =>
    {
        cmd.Parameters.AddWithValue("@Id", equipment.Id);
        cmd.Parameters.AddWithValue("@CharacterId", equipment.CharacterId);
        cmd.Parameters.AddWithValue("@ItemId", equipment.ItemId);
        cmd.Parameters.AddWithValue("@Slot", equipment.Slot);
    });

    _entityIds["Equipment"] = equipments.Select(e => e.Id).ToList();
}
```

Рисунок 3.3 – Генерація спорядження для персонажів (рисунок виконаний самостійно)

Використання такої складної логіки на етапі заповнення бази даних полегшить розробку запитів для тестування міграції, а також забезпечить реалістичність експериментальних умов. Зокрема, завдяки цьому згенеровані дані матимуть характеристики, максимально наближені до даних, що зустрічаються у реальних системах. Це, у свою чергу, дозволить більш точно оцінити ефективність алгоритмів міграції та їх здатність працювати з реальними сценаріями, мінімізуючи ризики некоректної роботи в практичному застосуванні.

Для доступу до даних використовувалася бібліотека Microsoft.Data.SqlClient. Для обробки заданих параметрів генерації було використано утиліту CommandLineParser.

Отже, розроблений додаток відповідає вимогам, визначеним на етапі проектування.

3.2 Реалізація алгоритмів міграції

Першим етапом міграції є аналіз вхідної структури реляційної бази даних. Результатом цього етапу є повна інформація про схему базу даних, включаючи первинні та зовнішні ключі. Ця інформація зберігається у відповідних моделях.

```
public class TableSchema
{
    public string Name { get; set; }
    public List<ColumnSchema> Columns { get; set; } = [];
    public List<string> PrimaryKeys { get; set; } = [];
    public List<ForeignKeySchema> ForeignKeys { get; set; } = [];
}
```

Отримання інформації про таблиці є незалежними операціями, тому вони можуть бути виконані паралельно. Експериментальним шляхом було виявлено, що паралелізацію доцільно використовувати при кількості таблиць більшої ніж 10, інакше витрати на керування потоками перевищують вигоду від паралельного виконання.

Паралельну версію алгоритму отримання схеми бази даних зображено на рисунку 3.4.

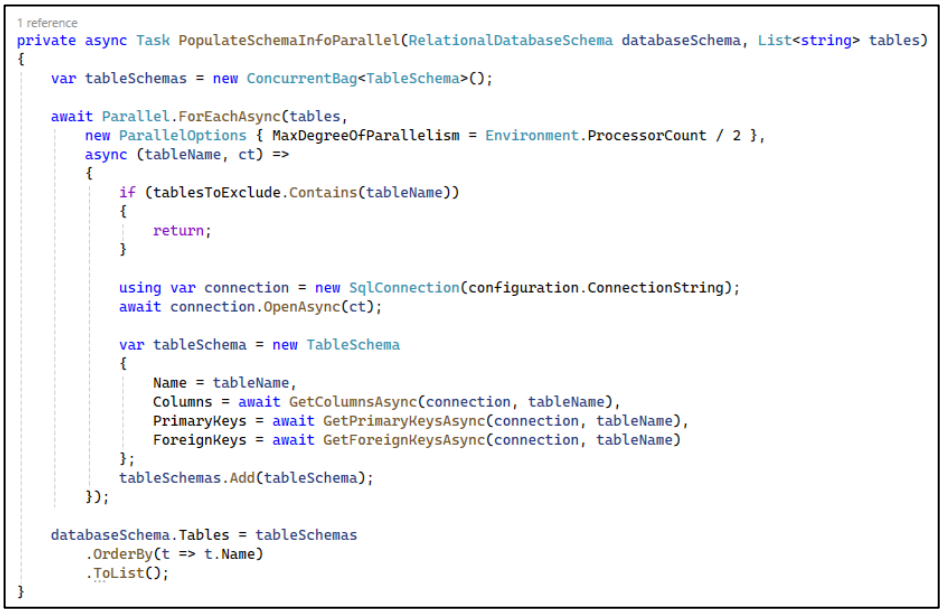
Першим кроком є отримання списку таблиць. Для цього виконується наступний SQL запит.

```
const string query = @"
    SELECT TABLE_NAME
    FROM INFORMATION_SCHEMA.TABLES
    WHERE TABLE_TYPE = 'BASE TABLE'";
```

Після цього для кожної таблиці отримується інформація про її стовбці (назва, тип даних).

```
const string query = @"
```

```
    SELECT
        COLUMN_NAME,
        DATA_TYPE,
        CHARACTER_MAXIMUM_LENGTH,
        IS_NULLABLE,
        COLUMN_DEFAULT
    FROM INFORMATION_SCHEMA.COLUMNS
    WHERE TABLE_NAME = @TableName
    ORDER BY ORDINAL_POSITION";
```



```
1 reference
private async Task PopulateSchemaInfoParallel(RelationalDatabaseSchema databaseSchema, List<string> tables)
{
    var tableSchemas = new ConcurrentBag<TableSchema>();

    await Parallel.ForEachAsync(tables,
        new ParallelOptions { MaxDegreeOfParallelism = Environment.ProcessorCount / 2 },
        async (tableName, ct) =>
        {
            if (tablesToExclude.Contains(tableName))
            {
                return;
            }

            using var connection = new SqlConnection(configuration.ConnectionString);
            await connection.OpenAsync(ct);

            var tableSchema = new TableSchema
            {
                Name = tableName,
                Columns = await GetColumnsAsync(connection, tableName),
                PrimaryKeys = await GetPrimaryKeysAsync(connection, tableName),
                ForeignKeys = await GetForeignKeysAsync(connection, tableName)
            };
            tableSchemas.Add(tableSchema);
        });

    databaseSchema.Tables = tableSchemas
        .OrderBy(t => t.Name)
        .ToList();
}
```

Рисунок 3.4 – Паралельне отримання схеми реляційної бази даних (рисунок виконаний самостійно)

Для отримання інформації про первинні ключі було використано наступний SQL запит.

```
const string query = @"
    SELECT
        Col.COLUMN_NAME
    FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS Tab
    JOIN INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE Col
        ON Col.CONSTRAINT_NAME = Tab.CONSTRAINT_NAME
    WHERE Tab.CONSTRAINT_TYPE = 'PRIMARY KEY'
        AND Tab.TABLE_NAME = @TableName";
```

Під час аналізу схеми бази даних також збирається інформація про зовнішні ключі:

– назва колонки;

– назва таблиці, на яку посилається зовнішній ключ;

– назва колонки, на яку посилається зовнішній ключ.

На рисунку 3.5. наведено відповідний запит.

```
const string query = @"
    SELECT
        FK.name AS FKName,
        ParentSchema = SCHEMA_NAME(ParentTable.schema_id),
        ParentTable = ParentTable.name,
        ParentColumn = ParentCol.name,
        ReferencedSchema = SCHEMA_NAME(ReferencedTable.schema_id),
        ReferencedTable = ReferencedTable.name,
        ReferencedColumn = ReferencedCol.name
    FROM sys.foreign_keys FK
    INNER JOIN sys.tables ParentTable
        ON FK.parent_object_id = ParentTable.object_id
    INNER JOIN sys.tables ReferencedTable
        ON FK.referenced_object_id = ReferencedTable.object_id
    INNER JOIN sys.foreign_key_columns FKCols
        ON FK.object_id = FKCols.constraint_object_id
    INNER JOIN sys.columns ParentCol
        ON FKCols.parent_column_id = ParentCol.column_id
        AND FKCols.parent_object_id = ParentCol.object_id
    INNER JOIN sys.columns ReferencedCol
        ON FKCols.referenced_column_id = ReferencedCol.column_id
        AND FKCols.referenced_object_id = ReferencedCol.object_id
    WHERE ParentTable.name = @TableName";
```

Рисунок 3.5 – SQL запит для отримання інформації про зовнішні ключі (рисунок виконаний самостійно)

Розроблені методи міграції використовують отриману інформацію про схему реляційної бази даних для своєї роботи.

Всього було реалізовано три алгоритми для:

– методу Rel2Graph в класичній імплементації, яка була описана в статті;

– методу Rel2Graph, модифікований алгоритм, який використовує багатопоточність;

– розробленого покращеного методу міграції.

Першим кроком метод Rel2Graph розподіляє таблиці на таблиці зв’язування та таблиці сутностей за принципом, описаним в розділі 2. Нижче наведено відповідний код.

```
        private static bool IsEntityTable(TableSchema table)
        {
            return
                table.ForeignKeys.Count == 0 ||
                table.ForeignKeys.Count == 1 || table.ForeignKeys.Count > 
2 ||
                (table.ForeignKeys.Count == 2 && table.PrimaryKeys.Count 
== 1);
        }

        private static bool IsRelationshipTable(TableSchema table)
        {
            if (table.ForeignKeys.Count != 2)
                return false;
            return
                table.PrimaryKeys.Count != 1 ||
                table.PrimaryKeys.All(pk => table.ForeignKeys.Any(fk => 
fk.ColumnName == pk));
        }
```

Після цього для кожного рядка кожної таблиці сутності створюється вузел в графовій базі даних. Для цього використовується наступний Cypher запит.

```
var cypher = $@"CREATE (n:{table.Name} $props) RETURN true";
```

Потім кожний рядок таблиці зв’язування перетворюється на ребро графа. Це досягається за допомогою запиту, наведеного нижче (див. рис. 3.6).

```
var cypher = $@"
    MATCH (source:{fk1.ReferencedTableName} {{{fk1.ReferencedColumnName}: $sourceId}})
    MATCH (target:{fk2.ReferencedTableName} {{{fk2.ReferencedColumnName}: $targetId}})
    CREATE (source)-[r:{table.Name} $props]->(target) RETURN true";
```

Рисунок 3.6 – Cypher запит для перетворення рядка таблиці зв’зування на ребро графа (рисунок виконаний самостійно)

Останнім кроком є створення додаткових ребер графа, які відповідають зовнішнім ключам таблиць сутностей (див. рис. 3.7). Такі ребра не мають властивостей та слугують лише для відображення зв’язків між вузлами.

```
var cypher = $@"
    MATCH (source:{table.Name} {{{primaryIdColumnName}: $sourceId}})
    MATCH (target:{fk.ReferencedTableName} {{{fk.ReferencedColumnName}: $targetId}})
    CREATE (target)-[r:HAS_{fk.ReferencedTableName}_{table.Name}]->(source) RETURN true";
```

Рисунок 3.7 – Cypher запит для перетворення зовнішніх ключей на ребра графа (рисунок виконаний самостійно)

В класичній імплементації методу Rel2Graph всі дії виконуються послідовно, що може бути прийнятним для невеликих баз даних.

В модифікованому алгоритмі методу Rel2Graph були зроблені певні покращення, а саме паралелізація операцій створення ребер та зв’язків, використання оператору UNWIND.

Розглянемо реалізацію паралелізації. Результуючі таблиці сутностей чи зв’язування рівномірно розподіляються між налаштованою кількістю потоків. За замовчуванням кількість потоків дорівнює кількості логічних процесорів. Для синхронізації потоків використовуються семафори, які дозволяють контролювати максимальну кількість потоків, що можуть працювати з критичною секцією. В даному випадку семафори запобігають перевантаженню підключень до бази даних.

На рисунку 3.8 зображено метод для паралельної обробки таблиць.

Цей метод приймає на вхід функцію, яку необхідно паралельно виконати над рядками таблиці. Це дозволяє використовувати наведений метод як для обробки таблиць сутностей, так і таблиць зв’язування.

Такий самий підхід використовується і для паралельної обробки зовнішніх ключів таблиць сутностей.

Іншим значним покращенням алгоритму є використання оператору UNWIND.

Оператор UNWIND приймає список елементів та повертає кожен елемент як окремий рядок, тобто він розгортає колекцію в окремі рядки, які можуть бути оброблені в одній операції бази даних. Це надає наступні переваги:

– замість того, щоб виконувати окремі оператори CREATE для кожного вузла/зв’язку, надсилається один запит із кількома елементами;
– для кількох операцій потрібна лише одна транзакція, що значно зменшує накладні витрати;
– база даних обробляє кілька операцій за один раз, використовуючи внутрішню оптимізацію.

```
2 references
private async Task ProcessTablesInParallel<T>(
    List<T> tables,
    Func<T, CancellationToken, Task> processor,
    CancellationToken cancellationToken)
{
    var tasks = new List<Task>();
    var partitionCount = _maxConcurrentDbConnections;
    var partitionSize = (int)Math.Ceiling(tables.Count / (double)partitionCount);

    for (int i = 0; i < partitionCount; i++)
    {
        var startIndex = i * partitionSize;
        var count = Math.Min(partitionSize, tables.Count - startIndex);

        if (count <= 0) continue;

        var partition = tables.GetRange(startIndex, count);

        tasks.Add(Task.Run(async () =>
        {
            foreach (var item in partition)
            {
                if (cancellationToken.IsCancellationRequested)
                    break;

                await _dbConnectionSemaphore.WaitAsync(cancellationToken);
                try
                {
                    await processor(item, cancellationToken);
                }
                finally
                {
                    _dbConnectionSemaphore.Release();
                }
            }
        }, cancellationToken));
    }

    await Task.WhenAll(tasks);
}
```

Рисунок 3.8 – Метод для паралельної обробки таблиць (рисунок виконаний самостійно)

За замовчуванням кількість елементів, над якими буде виконуватися операція, дорівнює 1000. Тобто замість того, щоб робити 1000 окремих викликів бази даних для 1000 записів, виконується один виклик, який обробляє всі 1000 записів одночасно.

Запити Cypher були змінені, щоб підтримувати оператор UNWIND. Запит для створення вузлів наведено нижче.

```
var cypher = $@"
    UNWIND $items AS item
    CREATE (n:{tableName})
    SET n = item
    RETURN count(n)";
```

На рисунку 3.9 зображено оновлений запит для перетворення рядків таблиці зв’зування на ребра графа.

```
var cypher = $@"
    UNWIND $items AS item
    MATCH (source:{fk1.ReferencedTableName} {{{fk1.ReferencedColumnName}: item.sourceId}})
    MATCH (target:{fk2.ReferencedTableName} {{{fk2.ReferencedColumnName}: item.targetId}})
    CREATE (source)-[r:{relationshipType}]->(target)
    SET r = item.props
    RETURN count(r)";
```

Рисунок 3.9 – Оновлений Cypher запит для перетворення рядків таблиці зв’зування на ребра графа (рисунок виконаний самостійно)

Оновлений запит для перетворення зовнішніх ключей на ребра графа наведено на рисунку 3.10.

```
var cypher = $@"
    UNWIND $items AS item
    MATCH (source:{tableName} {{{primaryIdColumnName}: item.sourceId}})
    MATCH (target:{fk.ReferencedTableName} {{{fk.ReferencedColumnName}: item.targetId}})
    CREATE (target)-[r:{relationshipType}]->(source)
    RETURN count(r)";
```

Рисунок 3.10 – Оновлений Cypher запит для перетворення зовнішніх ключей на ребра графа (рисунок виконаний самостійно)

Зроблені модифікації дозволяють прискорити процес міграції в десятки разів.

Покращений метод використовує такі самі механізми підвищення ефективності, як і модифікований метод Rel2Graph.

Основною відмінністю покращеного методу є спосіб класифікації таблиць, який враховує аспекти, наведені в розділі 2. На рисунку 3.11 наведено код класифікації таблиць покращеним методом.

```
1 reference
private static (List<TableSchema> EntityTables, List<TableSchema> RelationshipTables) ClassifyTables(RelationalDatabaseSchema schema)
{
    var entityTables = new List<TableSchema>();
    var relationshipTables = new List<TableSchema>();

    foreach (var table in schema.Tables)
    {
        if (IsRelationshipTable(table))
            relationshipTables.Add(table);
        else
            entityTables.Add(table);
    }

    return (entityTables, relationshipTables);
}

1 reference
private static bool IsRelationshipTable(TableSchema table)
{
    if (table.ForeignKeys.Count != 2)
        return false;

    if (table.PrimaryKeys.Count == 1)
        return table.Columns.Count <= 4;

    if (table.PrimaryKeys.Count == 2)
        return table.Columns.Count <= 3;

    return false;
}
```

Рисунок 3.11 – Класифікація таблиць покращеним методом (рисунок виконаний самостійно)

Такий підхід до класифікації таблиць дозволяє значно зменшити кількість результуючих вузлів та зв’язків, що безпосередньо впливає на час міграції та розмір результуючої графової БД. Переваги покращеного методу будуть продемонстровані на етапі проведення експериментів.

3.3 Реалізація запитів для експериментального дослідження

Для перевірки коректності міграції необхідно реалізувати набір семантично еквівалентних запитів на мові SQL та Cypher. Результати цих запитів будуть використовуватися для оцінки якості міграції.

В розділі 3 було розроблено чотири запити для предметної області соціальної мережі та чотири запити для предметної області ігрового серверного додатку.

Розглянемо реалізацію запитів для предметної області соціальної мережі.

Першим запитом є отримання інформації про пости користувачів. На мові SQL запит виглядає наступним чином.

```
SELECT u.Username, p.Content, p.CreationDate
FROM [User] u
JOIN [Post] p ON u.UserId = p.UserId
WHERE u.Username LIKE 'A%'
ORDER BY p.CreationDate DESC;
```

Запити на мові Cypher будуть відрізнятися для баз даних мігрованих за допомогою методу Rel2Graph та покращеного методу.

Запит для методу Rel2Graph наведено нижче.

```
MATCH (u:User)
MATCH (p:Post)
MATCH (u)-[:HAS_User_Post]->(p)
WHERE u.Username STARTS WITH 'A'
RETURN u.Username, p.Content, p.CreationDate
ORDER BY p.CreationDate DESC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (u:User)-[r:HAS_User_Post]->(p:Post)
WHERE u.Username STARTS WITH 'A'
RETURN u.Username, p.Content, p.CreationDate
ORDER BY p.CreationDate DESC;
```

В даному випадку запити майже не змінилися.

Перейдемо до розгляду наступного запиту – отримання інформації про пости з актуальними хештегами.

SQL запит виглядає наступним чином.

```
SELECT h.Name, p.Content, hp.Relevance
FROM [Hashtag] h
JOIN [HashtagPost] hp ON h.HashtagId = hp.HashtagId
JOIN [Post] p ON p.PostId = hp.PostId
WHERE hp.Relevance > 0.7
ORDER BY hp.Relevance DESC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (h:Hashtag)
MATCH (p:Post)
```

```
MATCH (hp:HashtagPost)
MATCH (h)-[:HAS_Hashtag_HashtagPost]->(hp)
MATCH (p)-[:HAS_Post_HashtagPost]->(hp)
WHERE hp.Relevance > 0.7
RETURN h.Name, p.Content, hp.Relevance
ORDER BY hp.Relevance DESC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (p:Post)-[r:HashtagPost]->(h:Hashtag)
WHERE r.Relevance > 0.7
RETURN h.Name, p.Content, r.Relevance
ORDER BY r.Relevance DESC;
```

Бачимо, що запит для покращеного методу більш компактний через відсутність проміжних вузлів HashtagPost, адже ці сутності були перетворені на ребра.

Наступним запитом є аналіз навичок користувачів.

Нижче наведено SQL запит.

```
SELECT s.Name, s.Category, COUNT(us.UserProfileId) as UserCount,
AVG(us.Level) as AvgLevel
FROM [Skill] s
JOIN [UserSkill] us ON s.SkillId = us.SkillId
JOIN [UserProfile] up ON us.UserProfileId = up.UserProfileId
JOIN [User] u ON up.UserId = u.UserId
GROUP BY s.Name, s.Category
HAVING AVG(1.0 * us.Level) > 3
ORDER BY UserCount DESC, s.Name ASC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (s:Skill)
MATCH (us:UserSkill)
MATCH (up:UserProfile)
MATCH (u:User)
MATCH (s)-[:HAS_Skill_UserSkill]->(us)
MATCH (up)-[:HAS_UserProfile_UserSkill]->(us)
MATCH (u)-[:HAS_User_UserProfile]->(up)
WITH s.Name as Name, s.Category as Category, count(distinct up) as Us
erCount, avg(us.Level) as AvgLevel
WHERE AvgLevel > 3
RETURN Name, Category, UserCount, AvgLevel
ORDER BY UserCount DESC, Name ASC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (s:Skill)-[r:UserSkill]->(up:UserProfile)<-
[:HAS_User_UserProfile]-(u:User)
WITH s.Name as Name, s.Category as Category, count(up) as UserCount,
avg(r.Level) as AvgLevel
WHERE AvgLevel > 3
RETURN Name, Category, UserCount, AvgLevel
ORDER BY UserCount DESC, Name ASC;
```

В запиті для покращеного методу відсутні вузли UserSkill, тому що вони були перетворені на ребра. Завдяки цьому запит є більш зрозумілим та меншим за розміром.

Останнім запитом для предметної області соціальної мережі є пошук спільних друзів для певного користувача. Така функціональність використовується майже в усіх сучасних соціальних мережах.

Наведемо SQL запит.

```
SELECT
    u3.Username AS SuggestedFriend,
    COUNT(DISTINCT u2.UserId) AS CommonFriends
FROM [User] u1
JOIN [Friendship] f1 ON (u1.UserId = f1.InitiatorId OR u1.UserId =
f1.RequestedFriendId)
JOIN [User] u2 ON (u2.UserId = f1.InitiatorId OR u2.UserId =
f1.RequestedFriendId) AND u2.UserId <> u1.UserId
JOIN [Friendship] f2 ON (u2.UserId = f2.InitiatorId OR u2.UserId =
f2.RequestedFriendId)
JOIN [User] u3 ON (u3.UserId = f2.InitiatorId OR u3.UserId =
f2.RequestedFriendId) AND u3.UserId <> u2.UserId
WHERE
    u1.Username = 'Cecile52'
    AND u3.UserId <> u1.UserId
    AND f1.Status = 1
    AND f2.Status = 1
    AND NOT EXISTS (
        SELECT 1 FROM [Friendship] f3
        WHERE f3.Status = 1
        AND ((f3.InitiatorId = u1.UserId AND f3.RequestedFriendId =
u3.UserId)
        OR (f3.InitiatorId = u3.UserId AND f3.RequestedFriendId =
u1.UserId))
    )
GROUP BY u3.Username
ORDER BY SuggestedFriend DESC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (u:User {Username: 'Cecile52'})

MATCH (f1:Friendship)
WHERE f1.Status = 1

MATCH (friend:User)
WHERE friend <> u

MATCH (u)-[:HAS_User_Friendship]->(f1)
MATCH (friend)-[:HAS_User_Friendship]->(f1)

MATCH (f2:Friendship)
WHERE f2.Status = 1

MATCH (fof:User)
WHERE fof <> friend AND fof <> u

MATCH (friend)-[:HAS_User_Friendship]->(f2)
MATCH (fof)-[:HAS_User_Friendship]->(f2)

WHERE NOT EXISTS {
    MATCH (u)-[:HAS_User_Friendship]->(f3:Friendship {Status: 1})<-
[:HAS_User_Friendship]-(fof)
}

RETURN fof.Username AS SuggestedFriend, COUNT(DISTINCT friend) AS Com
monFriends
ORDER BY SuggestedFriend DESC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (u:User {Username: 'Cecile52'})-[:Friendship {Status: 1}]-
(friend:User)-[:Friendship {Status: 1}]-(fof:User)
WHERE NOT (u)-[:Friendship {Status: 1}]-(fof) AND u <> fof
RETURN fof.Username AS SuggestedFriend, COUNT(DISTINCT friend) AS Com
monFriends
ORDER BY SuggestedFriend DESC;
```

На цьому прикладі дуже гарно видно переваги зберігання складних зв’язків в правильно спроектованих графових базах даних. Запит для покращеного методу є набагато меншим, простішим та інтуїтивно зрозумілим у порівнянні з запитами SQL та для методу Rel2Graph. Це пояснюється тим, що відносини Friendship зберігаються у вигляді ребер графа в базі даних, що була мігрована покращеним методом.

Тепер розглянемо реалізацію запитів для предметної області ігрового серверного додатку.

Першим запитом є отримання інформації про екіпіровку персонажа. На мові SQL запит виглядає наступним чином.

```
SELECT
    c.Name AS CharacterName,
    i.Name AS ItemName,
    e.Slot AS EquipmentSlot,
    a.Name AS AttributeName,
    ia.Value AS AttributeValue
FROM
    [Character] c
    JOIN Equipment e ON c.Id = e.CharacterId
    JOIN Item i ON e.ItemId = i.Id
    JOIN ItemAttribute ia ON i.Id = ia.ItemId
    JOIN Attribute a ON ia.AttributeId = a.Id
WHERE
    c.Id = 1
ORDER BY EquipmentSlot DESC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (c:Character {Id: 1})
MATCH (e:Equipment)<-[:HAS_Character_Equipment]-(c)
MATCH (i:Item)-[:HAS_Item_Equipment]->(e)
MATCH (a:Attribute)-[:HAS_Attribute_ItemAttribute]-
>(ia:ItemAttribute)<-[:HAS_Item_ItemAttribute]->(i)
RETURN
    c.Name AS CharacterName,
    i.Name AS ItemName,
    e.Slot AS EquipmentSlot,
    a.Name AS AttributeName,
    ia.Value AS AttributeValue
ORDER BY EquipmentSlot DESC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (c:Character {Id: 1})
MATCH (i:Item)<-[e:Equipment]-(c)
MATCH (a:Attribute)-[ia:ItemAttribute]->(i)
RETURN
    c.Name AS CharacterName,
    i.Name AS ItemName,
    e.Slot AS EquipmentSlot,
    a.Name AS AttributeName,
    ia.Value AS AttributeValue
```

```
ORDER BY EquipmentSlot DESC;
```

В покращеному методі вузли Equipment та ItemAttribute були перетворені на ребра, що дозволило зменшити кількість операцій MATCH.

Перейдемо до розгляду наступного запиту – аналіз предметів інвентаря персонажів.

SQL запит виглядає наступним чином.

```
SELECT
    u.Nickname AS PlayerName,
    c.Name AS CharacterName,
    i.Name AS ItemName,
    SUM(inv.Count) AS TotalItems
FROM
    [User] u
    JOIN [Character] c ON u.Id = c.UserId
    JOIN InventoryItem inv ON c.Id = inv.CharacterId
    JOIN Item i ON inv.ItemId = i.Id
WHERE
    i.Name = 'Generic Tasty Metal Tuna'
GROUP BY
    u.Nickname, c.Name, i.Name
ORDER BY
    TotalItems DESC, PlayerName ASC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (u:User)-[:HAS_User_Character]->(c:Character)
MATCH (c)-[:HAS_Character_InventoryItem]->(inv:InventoryItem)<-
[:HAS_Item_InventoryItem]-(i:Item)
WHERE i.Name = 'Generic Tasty Metal Tuna'
RETURN
    u.Nickname AS PlayerName,
    c.Name AS CharacterName,
    i.Name AS ItemName,
    SUM(inv.Count) AS TotalItems
ORDER BY
    TotalItems DESC, PlayerName ASC;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (u:User)-[:HAS_User_Character]->(c:Character)
MATCH (i:Item {Name: 'Generic Tasty Metal Tuna'})<-
[inv:InventoryItem]-(c)
```

```
RETURN
    u.Nickname AS PlayerName,
    c.Name AS CharacterName,
    i.Name AS ItemName,
    SUM(inv.Count) AS TotalItems
ORDER BY
    TotalItems DESC, PlayerName ASC;
```

В покращеному методі вузли InventoryItem були перетворені на ребра, що дозволило зменшити розмір та складність запиту.

Наступним запитом є комплексний аналіз квестів (перегляд інформації про нагороди за квести).

Нижче наведено SQL запит.

```
SELECT
    c.Name AS CharacterName,
    q.Name AS QuestName,
    qr.Name AS QuestRewardName,
    a.Name AS AttributeName,
    ia.Value AS AttributeValue
FROM
    [Character] c
    JOIN QuestProgress qp ON c.Id = qp.CharacterId
    JOIN Quest q ON qp.QuestId = q.Id
    JOIN QuestReward qr_link ON q.Id = qr_link.QuestId
    JOIN Item qr ON qr_link.ItemId = qr.Id
    JOIN ItemAttribute ia ON qr.Id = ia.ItemId
    JOIN Attribute a ON ia.AttributeId = a.Id
WHERE
    a.Name = 'Enchanting'
    AND ia.Value > 10
    AND qp.CurrentStage = 'Stage 1: Defeat';
```

Запит для методу Rel2Graph наведено нижче.

```
    MATCH (c:Character)-[:HAS_Character_QuestProgress]-
>(qp:QuestProgress)<-[:HAS_Quest_QuestProgress]-(q:Quest)
    MATCH (q)-[:HAS_Quest_QuestReward]->(qr_link:QuestReward)<-
[:HAS_Item_QuestReward]-(qr:Item)
    MATCH (qr)-[:HAS_Item_ItemAttribute]-(ia:ItemAttribute)<-
[:HAS_Attribute_ItemAttribute]-(a:Attribute {Name: 'Enchanting'})
    WHERE
        ia.Value > 10
        AND qp.CurrentStage = 'Stage 1: Defeat'
    RETURN
        c.Name AS CharacterName,
        q.Name AS QuestName,
        qr.Name AS QuestRewardName,
```

```
    a.Name AS AttributeName,
    ia.Value AS AttributeValue;
```

Тепер наведемо запит для покращеного методу.

```
MATCH (c:Character)-[:HAS_Character_QuestProgress]-
>(qp:QuestProgress)<-[:HAS_Quest_QuestProgress]-(q:Quest)
MATCH (q)<-[qr_link:QuestReward]-(qr:Item)
MATCH (qr)<-[ia:ItemAttribute]-(a:Attribute {Name: 'Enchanting'})
WHERE
    ia.Value > 10
    AND qp.CurrentStage = 'Stage 1: Defeat'
RETURN
    c.Name AS CharacterName,
    q.Name AS QuestName,
    qr.Name AS QuestRewardName,
    a.Name AS AttributeName,
    ia.Value AS AttributeValue;
```

В покращеному методі вузли QuestReward та ItemAttribute були перетворені на ребра графу.

Останнім запитом для предметної області ігрового серверного додатку є отримання інформації про лут персонажа.

Наведемо SQL запит.

```
SELECT
    c.Name AS CharacterName,
    l.DropDate,
    i.Name AS ItemName,
    di.Quantity
FROM
    [Character] c
    JOIN Loot l ON c.Id = l.CharacterId
    JOIN DroppedItem di ON l.Id = di.LootId
    JOIN Item i ON di.ItemId = i.Id
WHERE
    c.Id = 8
ORDER BY
    l.DropDate DESC;
```

Запит для методу Rel2Graph наведено нижче.

```
MATCH (c:Character {Id: 8})-[:HAS_Character_Loot]->(l:Loot)
```

```
    MATCH (l)-[:HAS_Loot_DroppedItem]->(di:DroppedItem)<-
[:HAS_Item_DroppedItem]-(i:Item)
    RETURN
        c.Name AS CharacterName,
        l.DropDate,
        i.Name AS ItemName,
        di.Quantity
    ORDER BY
        l.DropDate DESC;
```

Тепер наведемо запит для покращеного методу.

```
    MATCH (c:Character {Id: 8})-[:HAS_Character_Loot]->(l:Loot)
    MATCH (i:Item)-[di:DroppedItem]->(l)
    RETURN
        c.Name AS CharacterName,
        l.DropDate,
        i.Name AS ItemName,
        di.Quantity
    ORDER BY
        l.DropDate DESC;
```

В покращеному методі вузли DroppedItem були перетворені на ребра графу.

Незважаючи на більш компактну структуру, удосконалений підхід повністю зберігає семантичну еквівалентність запитів, забезпечуючи ідентичні результати.

Отже, реалізовані запити відповідають вимогам, визначеним на етапі розробки запитів для експериментального дослідження.

4 ОПИС ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

4.1 Проведення експериментальних досліджень

З використанням метрик, описаних в розділі 2.7, експериментальним шляхом було порівняно алгоритми Rel2Graph в класичній імплементації, оптимізований алгоритм Rel2Graph, який використовує багатопоточність, та розроблений алгоритм для Neo4j GraphMigr8.

Додатково для аналізу ефективності методів міграції проводилася оцінка продуктивності роботи з отриманими графовими моделями шляхом виконання еквівалентних запитів на графових структурах, створених різними алгоритмами міграції.

Для проведення серії експериментів було розглянуто п’ять найбільш популярних для використання графових БД предметних областей, в тому числі області соціальних мереж та ігрових серверних систем, для яких було розроблено відповідні БД:

- Ecommerce, що відповідає предметній області електронної комерції (9500098 рядків);
- SocialNetwork, що відповідає предметній області соціальних мереж (6744 рядків);
- GameDB, що відповідає предметній області ігрових серверних систем (7885 рядків);
- TravelingDb, що відповідає предметній області проведення подорожей (136 рядків);
- ReportDb, що відповідає предметній області волонтерської діяльності (45 рядків);
- SportDb, що відповідає предметній області спортивних змагань (70 рядків).

Розглянемо результати експериментів на БД середнього розміру, які стосувалися такого важливого показника ефективності, як час, що необхідний для перетворення реляційної БД у графову (див. табл. 4.1).

Таблиця 4.1 – Час міграції даних для БД середнього розміру (таблиця виконана самостійно)

Алгоритм	Час міграції SocialNetwork (с)	Час міграції GameDB (с)
Rel2Graph	213.27	202.51
Rel2Graph оптимізований	8.71	15.24
GraphMigr8	7.93	5.5

Оптимізований метод Rel2Graph та GraphMigr8 для Neo4J метод демонструють значне скорочення часу міграції для обох наборів даних завдяки оптимізації за рахунок використання багатопоточності. Метод GraphMigr8 демонструє найкращі показники через меншу кількість результуючих вузлів та зв’язків, що він утворює під час міграції.

Не менш важливим є замір використання системних ресурсів під час роботи алгоритмів (див. табл. 4.2).

Таблиця 4.2 – Використання системних ресурсів для БД середнього розміру (таблиця виконана самостійно)

Алгоритм	Використання оперативної пам’яті SocialNetwork (МБ)	Використання оперативної пам’яті GameDB (МБ)	Використання процесору SocialNetwork (%)	Використання процесору GameDB (%)
Rel2Graph	6.7	9.83	14.31	13.99
Rel2Graph оптимізований	15.19	11.08	3.79	2.39
GraphMigr8	11.29	6.47	2.99	3.16

Хоча алгоритм GraphMigr8 не показує очевидної переваги з точки зору використання оперативної пам'яті, але він значно знижує навантаження на процесор, що робить його більш ефективним стосовно використання системних ресурсів, а особливо для систем, де CPU є обмеженим ресурсом.

Наведемо отримані результати для різних обсягів вхідних реляційних БД. Великою БД будемо вважати базу даних Ecommerce, середніми – SocialNetwork та GameDB, та малими – TravelingDb, ReportDb, SportDb. Для середніх та малих БД наведемо усереднені значення. Отримані метрики наведені в таблиці 4.3.

Таблиця 4.3 – Результати міграції для БД різного об’єму (таблиця виконана самостійно)

Розмір БД	Алгоритм	Час міграції (с)	Використання оперативної пам’яті (МБ)	Використання процесору (%)
Малий (до 100 рядків)	Rel2Graph	8.6	9.6	6.5
	Rel2Graph оптимізований	4.67	2.08	3.8
	GraphMigr8	3.51	1.3	1.88
Середній (7300 рядків)	Rel2Graph	207.89	8.23	14.15
	Rel2Graph оптимізований	11.98	13.14	3.09
	GraphMigr8	6.715	8.88	3.08
Великий (9500098 рядків)	Rel2Graph	10829.44	91.1	26.04
	Rel2Graph оптимізований	433.21	52.99	7.44
	GraphMigr8	427.84	42.2	7.07

Перейдемо до розгляду експериментів щодо визначення семантичної цілісності даних.

Було розроблено декілька запитів на мовах SQL та Cypher (офіційній мові запитів Neo4j) та виконано їх на реляційній БД під керівництвом MS SQL Server та графовій БД під керівництвом Neo4j. Результати запитів порівнювалися автоматизовано за допомогою розробленого програмного забезпечення.

Для перевірки семантичної еквівалентності даних використовувалися статистичні запити на підрахунок кількості сутностей різного типу, більш деталізовані запити на фільтрацію сутностей по різним атрибутам, а також запити на перевірку зв'язності сутностей, що описані у розділі 2.8.

Реалізовані запити, описані у розділі 3.3, повернули однакову кількість записів з ідентичними значеннями атрибутів, тож можна вважати, що результуючі мігровані дані є цілісними.

Метрики продуктивності отриманих графових БД за результатами виконання цих запитів зведені в таблиці 4.4.

Таблиця 4.4 – Результати виконання розроблених запитів (таблиця виконана самостійно)

База даних	Запит	Метод	Оперативна пам'ять (байти)	Час (мс)	CPU (%)	DB hits
SocialNetwork	запит №1	Rel2Graph	4472	3	7	445
		GraphMigr8	4472	2	6.3	445
		Покращення	0%	-33.3%	-10.0%	0%
	запит №2	Rel2Graph	45000	7	10.2	5331
		GraphMigr8	40608	5	8	3257
		Покращення	-9.8%	-28.6%	-21.6%	-38.9%
	запит №3	Rel2Graph	13672	5	7.3	3669
		GraphMigr8	12944	3	7.9	2512
		Покращення	-5.3%	-40.0%	+8.2%	-31.4%
	запит	Rel2Graph	14352	3	11.7	998

Кінець таблиці 4.4

База даних	Запит	Метод	Оперативна пам’ять (байти)	Час (мс)	CPU (%)	DB hits
	№4	GraphMigr8	9592	2	7.9	426
		Покращення	-33.2%	-33.3%	-32.5%	-57.3%
GameDB	запит №1	Rel2Graph	2536	2	7.2	834
		GraphMigr8	2104	2	4.6	758
		Покращення	-17.0%	0%	-36.1%	-9.1%
	запит №2	Rel2Graph	8592	3	7.1	617
		GraphMigr8	8584	2	7.4	533
		Покращення	-0.1%	-33.3%	+4.2%	-13.6%
	запит №3	Rel2Graph	360	1	8.2	227
		GraphMigr8	344	1	6.2	181
		Покращення	-4.4%	0%	-24.4%	-20.3%
	запит №4	Rel2Graph	2256	2	6.2	718
		GraphMigr8	2256	1	6.8	682
		Покращення	0%	-50.0%	+9.7%	-5.0%

Метод GraphMigr8 демонструє значне покращення використання оперативної пам'яті та показує кращі результати за часом виконання у більшості випадків. В середньому, метод GraphMigr8 демонструє нижчу завантаженість процесора.

Однією з ключових переваг методу GraphMigr8 є отримання більш досконалої структури графа відповідно до кількості вершин та ребер графа. Кількість вузлів та зв’язків результуючих графових моделей наведено в таблиці 4.5.

Таблиця 4.5 – Кількість вузлів та зв’язків (таблиця виконана самостійно)

База даних	Метод	Кількість вузлів	Кількість зв’язків
SocialNetwork	Rel2Graph	6744	12388
	GraphMigr8	2850	8494
База даних	Метод	Кількість вузлів	Кількість зв’язків
	Покращення	-57.7%	-31.4%
GameDB	Rel2Graph	7885	12879
	GraphMigr8	2717	7711
	Покращення	-65.5%	-40.1%

Зменшення кількості вузлів та зв'язків при використанні методу міграції GraphMigr8 безпосередньо вплинуло на розмір бази даних. Розміри результуючих графових БД наведено в таблиці 4.6.

Таблиця 4.6 – Розмір бази даних (таблиця виконана самостійно)

База даних	Метод	Розмір (мб)
SocialNetwork	Rel2Graph	3.35
	GraphMigr8	2.42
	Покращення	-27.8%
GameDB	Rel2Graph	2.46
	GraphMigr8	1.67
	Покращення	-32.1%

Останнім типом експерименту є перевірка співпадіння автоматично мігрованої структури до спроектованої розробником. Для цього вручну було спроектовано графові БД для вхідних реляційних БД, описаних раніше. Наведемо назви таблиць для реляційних БД, щоб краще зрозуміти назви вершин та ребер в отриманих графових БД:

- TravelingDb – Achievement, Role, User, UserAchievement;
- ReportDb – HelpCategory, MeasurementUnit, Report, ReportDetail;

- SportDb – Command, Resolution, Task, Trainer;
- SocialNetwork – BlackList, Chat, Comment, Friendship, Hashtag, HashtagPost, Hobby, Like, Message, Post, Skill, User, UserChat, UserHobby, UserProfile, UserSkill;
- GameDb – Ability, Attribute, Character, CharacterAttribute, CharacterState, DroppedItem, Enemy, EnemyAttribute, Equipment, InventoryItem, Item, ItemAttribute, LoginHistory, Loot, NPC, Quest, QuestProgress, QuestReward, QuestStage, User, VictoryReward;
- Ecommerce – Basket, Category, Order, Product, Product_basket, Product_order, Product_property, Property, User.

Результати експерименту наведені в таблиці 4.7.

Таблиця 4.7 – Результати порівняння автоматично мігрованої структури зі спроектованою розробником (таблиця виконана самостійно)

База даних	Алгоритм	Кількість вузлів	Кількість зв’язків	Назви вузлів	Назви зв’язків
TravelingDb	Спроектовано вручну	44	112	Achievement, Role, User	HAS_ACHIEVEMENTS, HAS_ROLE
	Rel2Graph	136	204	Achievement, Role, User, UserAchievement	HAS_Achievement_UserAchievement, HAS_Role_User, HAS_User_UserAchievement
	GraphMigr8	44	112	Achievement, Role, User	HAS_Role_User, UserAchievement
SocialNetwork	Спроектовано вручну	4813	10457	Chat, Comment, Hashtag, Hobby, Like, Message, Post, Skill, User, UserProfile	HAS (4 зв’язки), SEND, LEAVES, PUTS, CREATES, FRIENDS_WITH, HAS_BLOCKED, CONTAINS (3 зв’язки)
	Rel2Graph	6744	12388	BlackList, Chat, Comment, Friendship, Hashtag, HashtagPost, Hobby, Like, Message, Post, Skill, User,	HAS_Chat_Message, HAS_Chat_UserChat, HAS_Hashtag_HashtagPost, HAS_Hobby_UserHobby, HAS_Post_Comment, HAS_Post_HashtagPost, HAS_Post_Like, HAS_Skill_UserSkill,

Продовження таблиці 4.7

База даних	Алгоритм	Кількість вузлів	Кількість зв’язків	Назви вузлів	Назви зв’язків
				UserChat,	HAS_UserProfile_UserHobby
				UserHobby, UserProfile, UserSkill	HAS_UserProfile_UserSkill, HAS_User_BlackList, HAS_User_Comment, HAS_User_Friendship, HAS_User_Like, HAS_User_Message, HAS_User_Post, HAS_User_UserChat, HAS_User_UserProfile
	GraphMigr8	2850	8494	Chat, Comment, Hashtag, Hobby, Message, Post, Skill, User, UserProfile	BlackList, Friendship, HAS_Chat_Message, HAS_Post_Comment, HAS_User_Comment, HAS_User_Message, HAS_User_Post, HAS_User_UserProfile, HashtagPost, Like, UserChat, UserHobby, UserSkill
GameDb	Спроектовано вручну	2581	7575	Ability, Attribute, Character, CharacterState, Enemy, Item, LoginHistory, Loot, NPC, Quest, QuestStage, User	LOGGED_IN, HAS_ATTRIBUTE (3 зв’язки), HAS_IN_INVENTORY, HAS_EQUIPPED, CONTAINS_REWARD (3 зв’язки), ACCOMPLISHES, OWNS, ISSUES, HAS (4 зв’язки)
	Rel2Graph	7885	12879	Ability, Attribute, Character, CharacterAttribute, CharacterState, DroppedItem,	HAS_Attribute_CharacterAttribute, HAS_Attribute_EnemyAttribute, HAS_Attribute_ ItemAttribute, HAS_Character_Ability,

Продовження таблиці 4.7

База даних	Алгоритм	Кількість вузлів	Кількість зв’язків	Назви вузлів	Назви зв’язків
				Enemy,	
				EnemyAttribute, Equipment, InventoryItem, Item, ItemAttribute, LoginHistory, Loot, NPC, Quest, QuestProgress, QuestReward, QuestStage, User, VictoryReward	HAS_Character_CharacterAttribute, HAS_Character_CharacterState, HAS_Character_Equipment, HAS_Character_InventoryItem, HAS_Character_Loot, HAS_Character_QuestProgress, HAS_Enemy_EnemyAttribute, HAS_Enemy_VictoryReward, HAS_Item_DroppedItem, HAS_Item_Equipment, HAS_Item_InventoryItem, HAS_Item_ItemAttribute, HAS_Item_QuestReward, HAS_Item_VictoryReward, HAS_Loot_DroppedItem, HAS_NPC_Quest, HAS_Quest_QuestProgress, HAS_Quest_QuestReward, HAS_Quest_QuestStage, HAS_User_Character, HAS_User_LoginHistory
	GraphMigr8	2717	7711	Ability, Attribute, Character, CharacterState, Enemy, Item, LoginHistory, Loot, NPC, Quest,	CharacterAttribute, DroppedItem, EnemyAttribute, Equipment, HAS_Character_Ability, HAS_Character_CharacterState, HAS_Character_Loot, HAS_Character_QuestProgress, HAS_NPC_Quest,

Кінець таблиці 4.7

База даних	Алгоритм	Кількість вузлів	Кількість зв’язків	Назви вузлів	Назви зв’язків
				QuestProgress, QuestStage, User	HAS_Quest_QuestProgress, HAS_Quest_QuestStage,
					HAS_User_Character, HAS_User_LoginHistory, InventoryItem, ItemAttribute, QuestReward, VictoryReward
Ecommerce	Спроектовано вручну	1800098	9400041	Basket, Category, Order, Product, Property, User	HAS_ORDER, HAS_BASKET, PLACED_IN, HAS_PROPERTY, ORDERED, HAS_PARENT
	Rel2Graph	1800098	9400041	Basket, Category, Order, Product, Property, User	HAS_Category_Category, HAS_Category_Product, HAS_User_Basket, HAS_User_Order, Product_basket, Product_order, Product_property
	GraphMigr8	1800098	9400041	Basket, Category, Order, Product, Property, User	HAS_Category_Category, HAS_Category_Product, HAS_User_Basket, HAS_User_Order, Product_basket, Product_order, Product_property

Аналіз результатів проведених експериментів наведено в розділі 5.

Таким чином, були проведені необхідні експерименти для порівняння досліджуваних алгоритмів міграції даних з реляційної БД в графову.

5 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

5.1 Порівняння продуктивності методів міграції

Після проведення експериментального дослідження необхідно проаналізувати отримані результати.

Наведемо графік порівняння часу міграції для різних обсягів вхідних реляційних БД (див. рис. 5.1, 5.2).

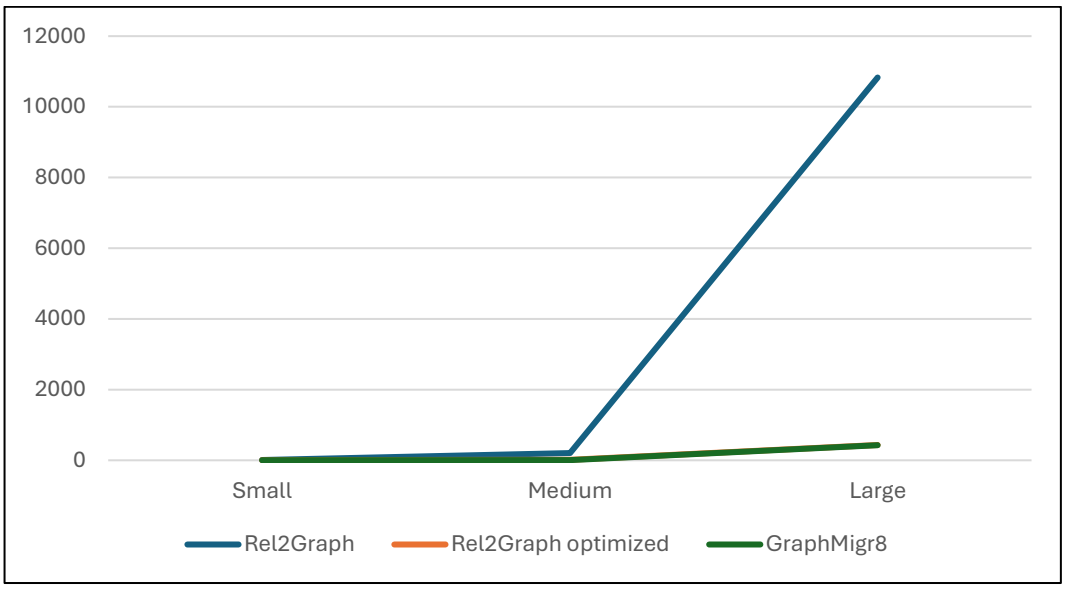


Рисунок 5.1 – Порівняння часу виконання алгоритмів для всіх розмірів вхідної БД (рисунок виконаний самостійно)

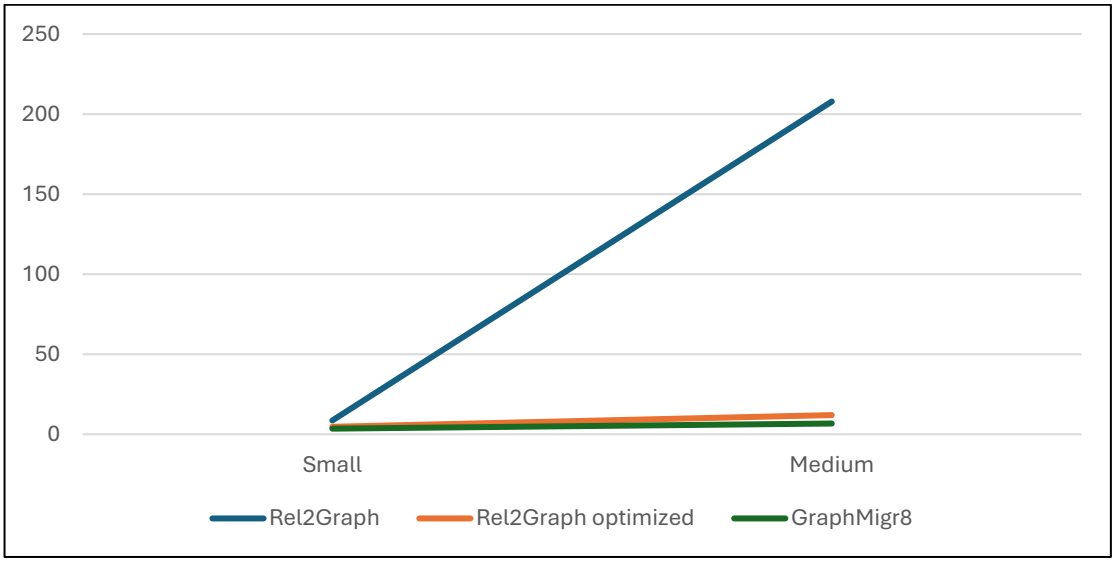


Рисунок 5.2 – Порівняння часу виконання алгоритмів для малих та середніх вхідних БД (рисунок виконаний самостійно)

Оригінальний Rel2Graph демонструє яскраво виражену експоненційну залежність від розміру БД (див. рис. 5.1). Перехід від середньої до великої БД спричиняє зростання часу міграції у понад 52 рази (з 207.89 с до 10829.44 с). Оптимізований Rel2Graph значно зменшує експоненційне зростання, хоча залежність все ще не є лінійною. Алгоритм GraphMigr8 показує найбільш наближену до лінійної залежність, що є оптимальним для масштабування. На великій БД оптимізований Rel2Graph та GraphMigr8 показали майже однаковий час міграції. Це пов’язано з тим, що предметна області Ecommerce не є оптимальною для зберігання у графовій БД . Реляційна схема цієї БД містить таблиці, які обома методами класифікуються як таблиці вершин, що не дозволяє задіяти сильні сторони методу GraphMigr8. Незважаючи на це, розроблений GraphMigr8 показав не гірші результати, ніж метод Rel2Graph.

Перейдемо до порівняння використання пам’яті під час роботи алгоритмів (див. рис. 5.3).

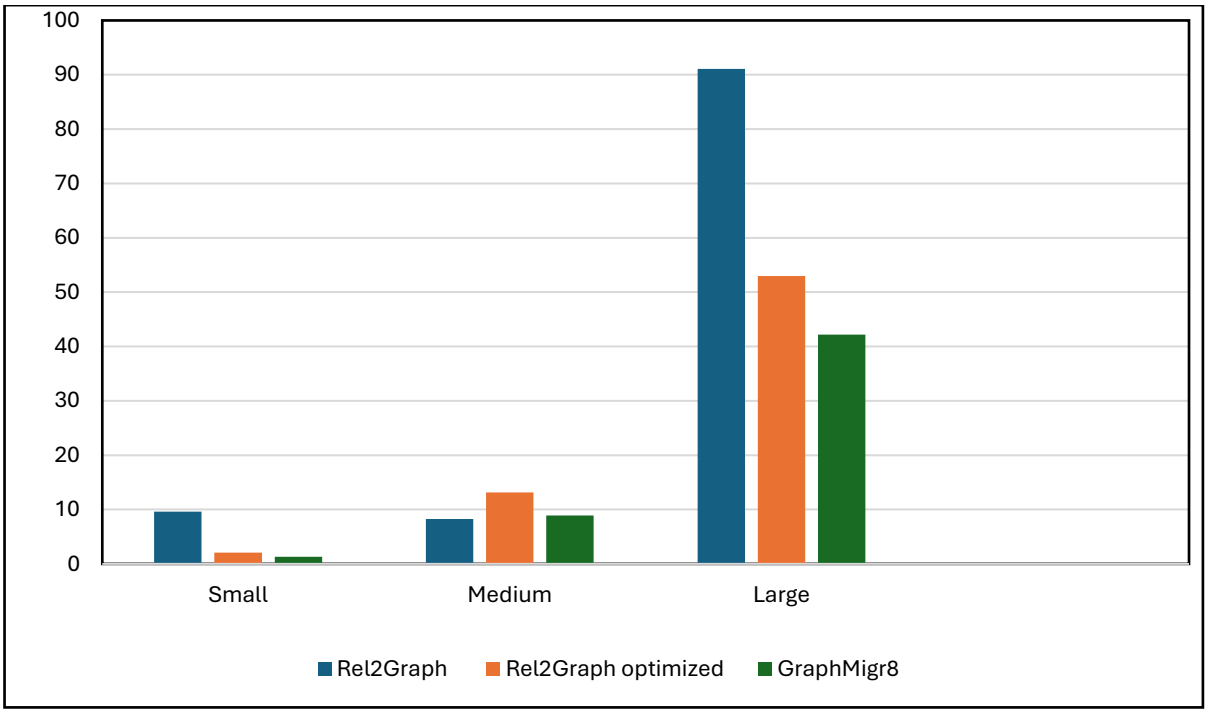


Рисунок 5.3 – Порівняння використання пам’яті під час роботи алгоритмів (рисунок виконаний самостійно)

Для малих БД алгоритм GraphMigr8 використовує на 86% менше пам'яті порівняно з оригінальним Rel2Graph. При збільшенні розміру БД спостерігається

зростання використання пам'яті в усіх алгоритмів, проте алгоритм GraphMigr8 завжди демонструє найнижчі показники.

Порівняємо завантаженість процесора під час роботи алгоритмів (див. рис. 5.4).

Оригінальний Rel2Graph найбільш інтенсивно використовує процесор, особливо при роботі з великими БД. Оптимізований Rel2Graph та GraphMigr8 демонструють приблизно однакове навантаження на процесор для середніх і великих БД. Алгоритм GraphMigr8 має найменше навантаження на процесор при роботі з малими БД.

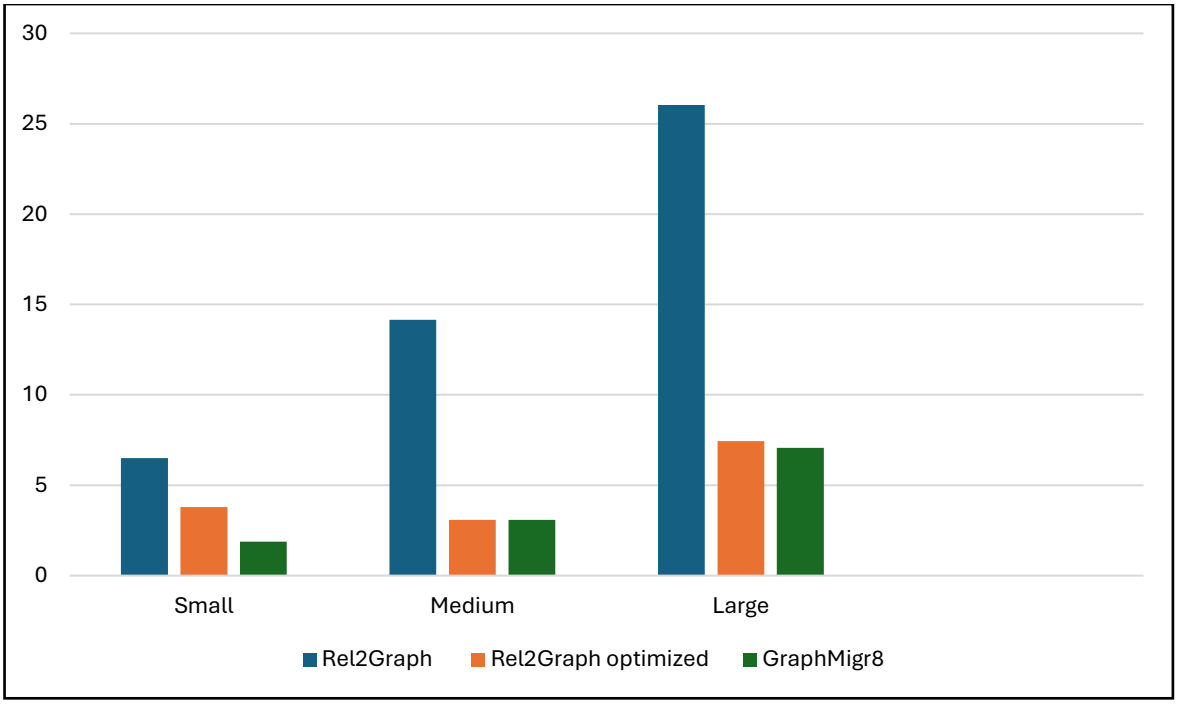


Рисунок 5.4 – Порівняння завантаженості процесора під час роботи алгоритмів (рисунок виконаний самостійно)

Проаналізуємо результати перевірки співпадіння автоматично мігрованої структури до спроектованої розробником.

Результати експериментів для БД малого розміру показали, що графові моделі, які були мігровані за допомогою методу GraphMigr8, мають більш ідентичну структуру до БД, що були спроєктовані вручну. В той час як графові моделі, що були мігровані за допомогою методу Rel2Graph, менш ідентичні до спроектованих вручну (приклад, БД TravelingDb).

БД середнього розміру для предметних областей соціальних мереж та ігрових серверних застосунків, які характеризуються високим ступенем взаємозв'язків між сутностями, є найбільш прийнятними кандидатами для використання графових БД та, відповідно, більш показовими об'єктами для проведення експериментальних досліджень. Для БД SocialNetwork метод GraphMigr8 продемонстрував більш компактну графову структуру на відміну від спроектованої вручну. Так, таблиця Like була перетворена на ребра між вершинами User та Post, що призвело до більш компактної та логічної структури графа. Для БД GameDB спроектована вручну графова структура виявилася більш компактною. Але метод GraphMigr8 знову показав кращі результати ніж Rel2Graph.

Для БД Ecommerce графові структури для всіх 3-х методів виявилися ідентичними, якщо не брати до уваги назви ребер. Це можна пояснити тим, що всі проміжні таблиці (таблиці зв’язування) в реляційні БД мають складений первинний ключ, частини якого одночасно є зовнішніми ключами, та кількість змістовних атрибутів не перевищує 2. В такому випадку методи Rel2Graph та GraphMigr8 створюють однакові графові структури. До того ж, слід зазначити, що розглянутий фрагмент області електронної комерції майже ніколи не проєктується розробниками в вигляді графової БД.

5.2 Розробка рекомендацій щодо використання методів міграції

Проведена серія експериментів демонструє, що БД з великою кількістю простих зв'язків (без додаткових змістовних атрибутів) отримують більшу користь від міграції до графової моделі через алгоритм GraphMigr8. Натомість, у системах електронної комерції, де зв'язки містять багато власних змістовних атрибутів, методи GraphMigr8 та Rel2Graph демонструють майже ідентичні результати. Це пояснює, чому метод показав різні результати на різних типах баз даних.

У випадках коли проміжні таблиці мають простий первинний ключ та два зовнішніх ключа, алгоритм GraphMigr8 буде повертати оптимальну графову структуру.

Отже, метод GraphMigr8 доречно використовувати в наступних сценаріях:

– бази даних з високою щільністю взаємозв'язків між сутностями;
– системи з великою кількістю простих зв'язків без додаткових змістовних атрибутів;
– предметні області, де проміжні таблиці мають простий первинний ключ та два зовнішніх ключі.

Метод Rel2Graph може бути використаний в наступних сценаріях:

– системи електронної комерції та фінансові системи, де зв'язки містять багато власних змістовних атрибутів;
– бази даних з складними проміжними таблицями, що мають додаткові атрибути крім зовнішніх ключів;
– системи, де важлива повна збереженість оригінальної структури даних.

Отже, проведені експерименти демонструють суттєві переваги методу GraphMigr8 до міграції реляційних БД у графові БД під керівництвом Neo4J порівняно з методом Rel2Graph, особливо для предметних областей з високою щільністю взаємозв'язків між сутностями, де графова модель даних є природно більш відповідною для представлення та обробки інформації.

Таким чином, проведені дослідження підтверджують ефективність запропонованого методу GraphMigr8 міграції реляційних БД в графові БД, демонструючи його переваги за ключовими показниками продуктивності.

Розроблений метод GraphMigr8 має певні обмеження, оскільки він розроблявся спеціально під міграцію даних до Neo4j, де у ребер є можливість зберігати атрибути. Проте не всі графові моделі підтримують цю функціональність. Між тим, метод GraphMigr8 без суттєвих змін підійде для міграції в такі графові СУБД, як Amazon Neptune, OrientDb, TigerGraph.

Натомість для таких СУБД як Apache Giraph або FlockDB, де концепція атрибутів ребер обмежена або відсутня, застосування методу GraphMigr8

потребуватиме модифікації. У цих випадках більш ефективним буде використання методу Rel2Graph.

В перспективі доцільним є проведення додаткових експериментальних досліджень з вищезгаданими СУБД для більш точної оцінки ефективності запропонованого методу в різних технологічних середовищах. Тим не менш, можна зробити прогноз, що якісні переваги методу GraphMigr8 збережуться порівняно з методом Rel2Graph, хоча ефективність кількісних показників може дещо змінитися. Ці відмінності можуть бути зумовленими особливостями реалізації відповідних СУБД, специфікою їх мов запитів, оптимізацією внутрішніх алгоритмів обробки даних та ефективністю роботи з різними типами структур.

ВИСНОВКИ

В результаті кваліфікаційної роботи було проаналізовано предметну галузь дослідження, досліджено існуючи підходи до міграції даних з реляційної в графову базу даних, також було спроєктовано реляційну та графову логічні моделі для обраних предметних областей.

Для проведення дослідження було обрано предметні області соціальних мереж, ігрових серверних систем, та інші. Було розроблено відповідні ER-діаграми, схеми реляційних БД та логічні моделі графових БД.

Під час дослідження існуючих методів міграції реляційних даних було виявлено певні недоліки методів, запропоновано та описано удосконалений метод GraphMigr8.

Запропоновано математичну модель міграції, яка дозволяє формалізувати процес перетворення реляційних структур у графову модель. Визначено базові принципи та алгоритми міграції, що забезпечують максимальне збереження семантики вихідних даних.

Також було проведено планування експериментального дослідження, в ході якого були визначені основі метрики для порівняння ефективності досліджуваних методів міграції даних.

Було спроєктовано та розроблено програмне забезпечення для проведення експериментального дослідження, розроблено та реалізовано запити для проведення експериментів, проведено експерименти. На основі результатів експериментів були розроблені рекомендації щодо використання досліджуваних методів міграції.

За результатами роботи були опубліковані тези «Трансформація структурованих даних в графову модель» на сімнадцяту міжнародну науково-технічну конференцію «Інформаційні технології та автоматизація – 2024» (див. додаток В) та прийнято до опублікування статтю «Методи міграції реляційних даних в графову модель Neo4J» в науковий журнал категорії Б «Сучасний стан наукових досліджень та технологій в промисловості» (див. додаток Г).

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Isaacson C. Understanding big data scalability: big data scalability series, part I. Pearson Education, Limited, 2014. 123 с.

2. DB-Engines Ranking. URL: https://db-engines.com/en/ranking (дата звернення: 14.10.2024).

3. Bechberger D., Perryman J. Graph Databases in Action: Examples in Gremlin. Manning Publications Co. LLC, 2020. 336 с.

4. Webber J., Robinson I., Eifrem E. Graph Databases: New Opportunities for Connected Data. O'Reilly Media, Incorporated, 2015. 238 с.

5. Graph Algorithms: Practical Examples in Apache Spark and Neo4j. O'Reilly Media, Incorporated, 2019. 300 с.

6. Anthapu R. Graph Data Processing with Cypher: A practical guide to building graph traversal queries using the Cypher syntax on Neo4j. Packt Publishing, 2022. 332 с.

7. Getting started with Amazon Neptune – Amazon Neptune. URL: https://docs.aws.amazon.com/neptune/latest/userguide/graph-get-started.html (дата звернення: 15.11.2024).

8. TigerGraph DB. URL: https://docs.tigergraph.com/tigergraph-server/4.1/intro/ (дата звернення: 15.11.2024).

9. OrientDB Manual. URL: https://orientdb.org/docs/3.0.x/ (дата звернення: 15.11.2024).

10. ArangoDB Documentation. URL: https://docs.arangodb.com/3.11/get-started/ (дата звернення: 15.11.2024).

11. Kuzochkina, A., Shirokopetleva, M., Dudar, Z. Analyzing and Comparison of NoSQL DBMS, International Scientific-Practical Conference on Problems of Infocommunications Science and Technology, PIC S and T 2018. Proceedings. 2019. P. 560–564. DOI 10.1109/INFOCOMMST.2018.8632133.

12. Neo4j architecture. URL: https://www.graphable.ai/blog/neo4j-performance/ (дата звернення: 15.11.2024).

13. 2019 Database Trends: SQL Vs. NoSQL - Top Databases.

URL: https://scalegrid.io/blog/2019-database-trends-sql-vs-nosql-top-databases-single-vs-multiple-database-use/ (дата звернення: 17.11.2024).

14. Solarz A., Szymczyk T. Oracle 19c, SQL Server 2019, Postgresql 12 and MySQL 8 database systems comparison. Journal of Computer Sciences Institute. 2020. T. 17. C. 373–378. URL: https://doi.org/10.35784/jcsi.2281 (дата звернення: 18.11.2024).

15. REL2GRAPH. URL: https://arxiv.org/pdf/2310.01080 (дата звернення: 20.11.2024).

16. Nan Z., Bai X. The study on data migration from relational database to graph database. Journal of Physics: Conference Series. 2019. T. 1345. C. 022061. URL: https://doi.org/10.1088/1742-6596/1345/2/022061 (дата звернення: 20.11.2024).

17. What is Entity Relationship Diagram (ERD)?. URL: https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/ (дата звернення: 22.11.2024).

18. Mapping from ER Model to Relational Model - GeeksforGeeks. URL: https://www.geeksforgeeks.org/mapping-from-er-model-to-relational-model/ (дата звернення: 25.11.2024).

19. Mazurova O., Syvolovskyi I., Syvolovska O. NOSQL database logic design methods for MONGODB and NEO4J. Innovative technologies and scientific solutions for industries, (2 (20)), 2022. P. 52–63. DOI: 10.30837/ITSSI.2022.20.052.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

11. Kuzochkina, A., Shirokopetleva, M., Dudar, Z. Analyzing and Comparison of NoSQL DBMS, International Scientific-Practical Conference on Problems of Infocommunications Science and Technology, PIC S and T 2018. Proceedings. 2019. P. 560–564. DOI 10.1109/INFOCOMMST.2018.8632133.

19. Mazurova O., Syvolovskyi I., Syvolovska O. NOSQL database logic design methods for MONGODB and NEO4J. Innovative technologies and scientific solutions for industries, (2 (20)), 2022. P. 52–63. DOI: 10.30837/ITSSI.2022.20.052.