Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
Кафедра Комп'ютерні інформаційні технології

Пояснювальна записка

до кваліфікаційної роботи
магістра

на тему: «Дослідження часової ефективності графових баз даних»
за освітньою програмою **12 Інженерія програмного забезпечення**
зі спеціальності: **121 Інженерія програмного забезпечення**

Виконав: студент групи ПЗ2221: _____ /Павло БАША /
(посада/підпис)

Керівник: _____ /Віктор ШИНКАРЕНКО/
(посада/підпис)

Нормоконтролер: _____ /Світлана ВОЛКОВА /
(посада/підпис)

Засвідчую, що у цій роботі немає
запозичень з праць інших авторів
без відповідних посилань.
Студент _____.

Дніпро – 2024 рік

Ministry of Education and Science of Ukraine

Ukrainian State University of Science and Technologies

Faculty Computer technologies and systems
Department Computer information technology

Explanatory Note

to Master's Thesis

on the topic: «Research on the time efficiency of graph databases»
according to educational curriculum **12 software engineering**
in the Speciality: **121 software engineering**

Done by the student of the group PZ2221: /Pavlo BASHA /

Scientific Supervisor: /Viktor SHYNKARENKO/

Normative controller: /Svitlana VOLKOVA /

Dnipro – 2024

Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет: Комп'ютерних технологій і систем
Кафедра: Комп’ютерні інформаційні технології
Рівень вищої освіти: магістр
Освітня програма: Інженерія програмного забезпечення
Спеціальність: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ____КІТ
_____ Вадим ГОРЯЧКІН
_____ січень 2023 р.

ЗАВДАННЯ

На кваліфікаційну роботу____Магістр_____

студенту Баші Павло Олександровичу

1. Тема дипломної роботи: Дослідження часової ефективності графових баз даних.

Керівник роботи: Шинкаренко Віктор Іванович

затверджені наказом 1113ст від 02.11.2022 року

2. Строк подання студентом роботи 12.01.2024 року

3. Вихідні дані до дипломної роботи:

_____.

4. Зміст пояснювальної записки (перелік питань до розробки):

4.1. Загальний огляд проблеми.

4.2. Обґрунтування напрямку досліджень.

4.3. Проектування і розробка ПЗ для досліджень.

4.4. Дослідження часової ефективності графових баз даних.

5. Перелік демонстраційного матеріалу:

5.1. презентація;

5.2. демонстраційне відео.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів	Примітка
1	Вступ	03.03.23 – 14.04.23	
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	15.04.23 – 22.08.23	
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження	23.08.23 – 18.10.23	
4	Постановка задачі, технічне завдання	19.10.23 – 20.10.23	30%
5	Техніко-економічні показники	21.10.23 – 07.11.23	
6	Розробка інструментальних засобів дослідження	08.11.23 – 13.11.23	
7	Виконання досліджень	14.11.23 – 18.11.23	60%
8	Оформлення тез доповідей	19.11.23 – 19.11.23	
9	Оформлення статті у фаховий журнал	19.11.23 – 22.11.23	
10	Оформлення пояснювальної записки	20.11.23 – 24.12.23	
11	Розробка демонстраційних матеріалів	24.12.23 – 25.12.23	100%
12	Подання кваліфікаційної роботи до кафедри	15.01.24	
13	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	21.01.24	

Студент: _____/Павло БАША/

Керівник роботи: _____/Віктор ШИНКАРЕНКО/

РЕФЕРАТ

Об’єктом даного дослідження є оптимальність графових баз даних у виконанні конкретних задач, порівняно с реляційними базами даних та іншими NO-SQL базами даних.

Предметом дослідження є процедури та оцінки результатів часової ефективності виконання пошуку по графовим базам даних.

Метою даної роботи є визначення доцільності застосування графових баз даних у задачах пошуку найближчих вершин та зв’язків між об’єктами, а саме можливість пришвидшити ці процеси завдяки правильному вибору типу бази даних під час проектування системи.

Результати та їх новизна: дослідження дозволять краще розуміти різницю між різними типами баз даних для оптимальнішого вибору цього типу під час створення архітектури програми, яка налаштована на здійсення певних задач пошуку та об’єкти якої мають достатню зв’язність між собою .

Пояснювальна записка складається зі вступу, 5 розділів, висновків, бібліографічного списку та 3 додатків:

– вступ – в даному розділі описується сутність розробки, її актуальність. Складається з 3 сторінок;

– у першому розділі висвітлено аналіз сучасного стану дослідження проблеми за науковими літературними джерелами, також проаналізовано сучасний стану програмно-апаратного забезпечення. Складається з 10 сторінок;

– другому розділі надано обґрунтування експериментального методу дослідження. Складається з 5 сторінок;

– у третьому розділі представлене проектування і розробка інструментального забезпечення для дослідження. Складається з 65 сторінок;

– четвертому розділі описані виконані дослідження. Складається з 9 сторінок;

– п‘ятому розділі розкриті питання охорона та безпеки праці. Складається з 20 сторінок;

– додатки містять технічне завдання, робочий проект та тези.

Ключові слова: графові бази даних, реляційні бази даних, часова ефективність.

ЗМІСТ

ВСТУП

Дослідження часової ефективності використання графових баз даних в порівняні з реляційними базами – це дуже сучасне і доцільне питання, так як у новіших прикладних задачах з'являються все більше ситуацій, де об'єкти пов'язані між собою та мають за мету часті запити до бази даних по отриманню цих зв'язків.

Продуктивність є однією з мотивів використовувати бази даних NoSQL замість традиційних баз даних SQL. З даними та запитами, які відповідають моделі даних, бази даних NoSQL можуть запропонувати значні переваги в продуктивності. Системи баз даних традиційної реляційної моделі та графової моделі NoSQL. У графовій моделі, яка є одним із чотирьох основних типів NoSQL, дані складаються з вузлів і ребер, і вона має власні переваги при обробці насичених зв'язками даних. У той час як у базах даних SQL може знадобитися об'єднати кілька таблиць для реляційного запиту, у графових базах даних реляційну інформацію можна запитувати шляхом навігації по графу.

Тема роботи: «Дослідження часової ефективності графових баз даних».

Актуальність роботи є процедури

Об'єктом дослідження є оптимальність графових баз даних у виконанні конкретних задач, порівняно с реляційними базами даних та іншими NO-SQL базами даних.

Предметом дослідження є процедури та оцінки результатів часової ефективності виконання пошуку по графовим базам даних.

Мета і завдання дослідження полягає у порівняння часової ефективності виборки тісно пов'язаних даних між собою, щоб порівняти їх с аналогічними часовими показниками звичайних реляційних баз даних або інших NO-SQL баз даних.

Поставлена мета зумовила необхідність вирішення наступного комплексу взаємопов'язаних завдань:

розробити програмне забезпечення для виміру часу виконання аналогічних запитів до баз даних різного типу;

дослідити ефективність та доцільність виконання запитів відповідно до кількості даних та їх зв’язку;

порівняти часову ефективність між отриманими результатами на різних типах баз даних.

Наукова новизна полягає у подальшому дослідженні можливостей графових баз даних та їх прикладного використання у певних задачах для оптимізації результату та покрашення часових показників роботи програмного забезпечення.

Практичне значення полягає у можливості підбору більш оптимального типу баз даних на основі проведених досліджень у прикладних задачах в реальному світі, що прискорить роботу по пошуку і вилученню даних для інтерактивних систем.

1 ЗАГАЛЬНИЙ ОГЛЯД ПРОБЛЕМИ

1.1 Призначення та область застосування

Графові бази даних набувають значущості в контексті збільшення складності та обсягу пов'язаних між собою даних. Застосування графових баз даних може бути важливим етапом для вищих навчальних закладів та інших галузей, які оптимально використовують представлення даних у вигляді графів. Система має надавати можливості для збору та аналізу даних, що взаємодіють в графовій структурі. Користувачі з базовими навичками в програмуванні та розумінням концепцій аналізу даних можуть ефективно використовувати цю систему.

Функціональне призначення графових баз даних полягає в створенні інтерактивних інструментів для аналізу та управління пов'язаними даними. Забезпечується функціональність для керування даними користувачів, контролю їхньої активності та відображення результатів аналізу попиту.

1.2 Постановка задачі

Метою є розробка програмного продукту для дослідження часової ефективності графових баз даних та їх порівняння з реляційними. Програмний продукт повинен враховувати особливості обробки пов'язаних між собою даних та враховувати фактори, які впливають на час виконання запитів.

Функціональні вимоги включають:

– проведення досліджень з часової ефективності різних графових баз даних та реляційних систем.

– забезпечення можливостей керування даними, визначенням параметрів для дослідження.

– виведення результатів досліджень у вигляді графіків та таблиць для зручності аналізу.

1.3 Огляд аналогів інструментарію для проведення досліджень

В сучасних умовах існує ряд програмних інструментів та методик, спрямованих на вимірювання часу виконання запитів до баз даних, що є

ключовим елементом для оцінки продуктивності систем управління базами даних (СУБД). Для цього використовуються різні підходи та інструменти, які можуть бути класифіковані за деякими категоріями.

У першу чергу, більшість СУБД надають вбудовані засоби для вимірювання часу виконання запитів. Наприклад, такі інструменти, як EXPLAIN і профілер запитів у MySQL, можуть служити для отримання детальної інформації щодо виконання конкретних запитів.

Для вимірювання продуктивності також використовують об'єктно-реляційні відображення (ORM), такі як SQLAlchemy для Python або Hibernate для Java, які надають функції вимірювання часу виконання запитів, спрощуючи роботу з базою даних.

Спеціалізовані бенчмаркінгові інструменти, такі як SysBench, HammerDB чи TPC Benchmarks, призначені для тестування продуктивності баз даних та визначення їх ефективності у визначених сценаріях використання. Важливо також зазначити, що деякі компанії розробляють власні інструменти для вимірювання продуктивності в конкретних використовуваних системах.

Нарешті, системи моніторингу, такі як Prometheus, Grafana чи DataDog, забезпечують можливість вимірювати та моніторити час виконання запитів у реальному часі, надаючи детальний огляд продуктивності баз даних у робочому середовищі. У виборі конкретного інструменту важливо враховувати конкретні вимоги проекту та особливості використовуваної бази даних.

1.4 Опис проблеми

На сучасному етапі розвитку інформаційних технологій у сфері обробки та зберігання даних, проблема вибору оптимальної системи управління базами даних (СУБД) стає ключовою, особливо в контексті порівняння реляційних та графових баз даних.

Реляційні бази даних традиційно використовуються для зберігання та обробки структурованих даних, а також виконання складних операцій SQL-запитів. Однак, зі зростанням обсягу пов'язаних між собою даних, реляційні СУБД можуть виявлятися неоптимальними, спричиняючи питання щодо

ефективності та продуктивності при обробці глибоких та складних зв'язків між об'єктами.

Графові бази даних визначають новий підхід до зберігання та опрацювання даних, оснований на представленні їх у вигляді графів. Це стає особливо важливим у випадках, коли важлива інформація взаємопов'язана та вимагає глибшого аналізу залежностей між елементами даних. Однак, вибір графової СУБД також супроводжується викликами, зокрема, визначенням оптимальності їхнього використання в конкретних сценаріях та у порівнянні з традиційними реляційними рішеннями.

Основна проблема полягає у здатності СУБД ефективно обробляти та взаємодіяти з великим обсягом пов'язаних даних при збереженні стабільної часової ефективності. Отже, вибір між реляційними та графовими базами даних стає стратегічним завданням, що вимагає ретельного дослідження та порівняння для досягнення оптимальних результатів у конкретних умовах використання.

Однією з ключових причин вибору графових баз даних в ряді сценаріїв є їхній інтуїтивно зрозумілий та ефективний спосіб моделювання та роботи з взаємозалежними даними. Графові бази даних відображають реальні зв'язки та взаємодії між об'єктами, надаючи зручний інструмент для вирішення завдань, де важливо розуміти структуру та взаємозв'язки між елементами інформації. Це особливо корисно в галузях, де аналіз мережевих взаємодій або залежностей між об'єктами є важливим етапом обробки даних.

З іншого боку, реляційні бази даних, винятково корисні для зберігання структурованих даних та виконання складних SQL-операцій, можуть виявитися менш ефективними при роботі з великим обсягом взаємозалежних даних. Проблема полягає в тому, що реляційні моделі не завжди ефективно відображають складні зв'язки, які часто трапляються в реальних сценаріях, і це може призвести до недооцінки ефективності та ускладнення операцій обробки даних.

Особливо помітно проблему часової неефективності в реляційних базах даних при роботі з глибокими та розгалуженими зв'язками. Запити, які вимагають

рекурсивного або множинного обходу графу, можуть призводити до значного зростання часу виконання в реляційних моделях, оскільки вони не є природньо спрямованими на обробку таких структур. Це може призвести до витрат часу та ресурсів на операції, які графові бази даних можуть ефективно оптимізувати за рахунок своєї специфічної структури.

1.5 Огляд програмних аналогів

Для роботи з будь-якими типами баз даних існують системи управління базами даних які мають назву Системи Управління Базою Даних (СУБД). Тож, для виконання наших досліджень спершу треба ознайомитись з функціоналом використання подібних систем, програмні інтерфейси яких ми будемо використовувати. Це не зовсім аналоги, але саме API протоколи, які будуть використовуватись для роботи з базами даних. Далі будуть наведені приклади інтерфейсів готових систем (СУБД), які пропонуються користувачу для роботи з певними типами баз даних.

1.5.1 Функціональні можливості СУБД «Sones GraphDB»

Sones GraphDB — це база даних графів, розроблена німецькою компанією sones GmbH, яка була доступна з 2010 по 2012 рік. Її остання версія була випущена в травні 2011 року. Sones GmbH, яка базувалася в Ерфурті та Лейпцигу, була оголошена банкрутом 1 січня 2012 року. GraphDB була унікальною тим, що її дизайн базувався на зважених графіках. Версія з відкритим кодом була випущена в липні 2010 року. Комерційно доступна корпоративна версія пропонувала більш широкий спектр функцій. GraphDB було розроблено на мові програмування C# і працювало на Microsoft .NET Framework і на Mono з відкритим вихідним кодом.

GraphDB була доступний як програмне забезпечення як послуга (SaaS) на хмарній платформі Microsoft Azure Services Platform. GraphDB також була компонентом стека рішень з відкритим кодом.

GraphDB має безіндексну суміжність, що означало, що не потрібно було керувати глобальним індексом для зв’язків між вузлами/сутностями. Пов’язані об’єкти містили пряме посилання на їхні сусідні вузли.

База даних графів sones могла зберігати та отримувати неструктуровані властивості в будь-якому вузлі графа. Ідея також полягала в тому, щоб перенести неструктуровані дані в структуровані дані і навпаки. Структуровані дані можна динамічно розширювати з високою продуктивністю у вузлах і ребрах під час виконання. Додаткові властивості можна було легко ввести або видалити з типів вершин за короткий проміжок часу.

GraphDB використовував власну мову запитів GraphQL, яка була схожа на SQL. Його можна динамічно розширювати під час виконання за допомогою плагінів, таких як функції або агрегати.

GraphDB використовував об’єктно-орієнтовану концепцію, яка дозволила краще інтегрувати об’єктно-орієнтовані мови програмування.

Система управління графовими базами даних Sones GraphDB вирізняється рядом особливостей, які роблять її унікальною в своєму роді.

Sones GraphDB використовує технологію графової бази даних для зберігання і обробки великої кількості пов'язаних даних. Основні особливості включають:

графова модель даних: Ця база даних побудована на основі графової моделі, де дані представлені у вигляді вузлів та ребер, що відображають взаємозв'язки між об'єктами;

оптимізація для пов'язаних даних: Sones GraphDB спроектована для оптимальної роботи з великою кількістю взаємозалежних даних, забезпечуючи ефективний пошук та аналіз структури графа;

підтримка запитів: Система підтримує високопродуктивні запити для витягування даних з графової структури, дозволяючи виконувати різноманітні аналітичні операції.

Для роботи з Sones GraphDB розроблено низку інструментів та інтерфейсів для полегшення взаємодії та управління базою даних:

– «Sones GraphDB Console»: Графічний інтерфейс користувача, що дозволяє виконувати запити, вивчати структуру графа та моніторити роботу бази даних (рис. 1.1);

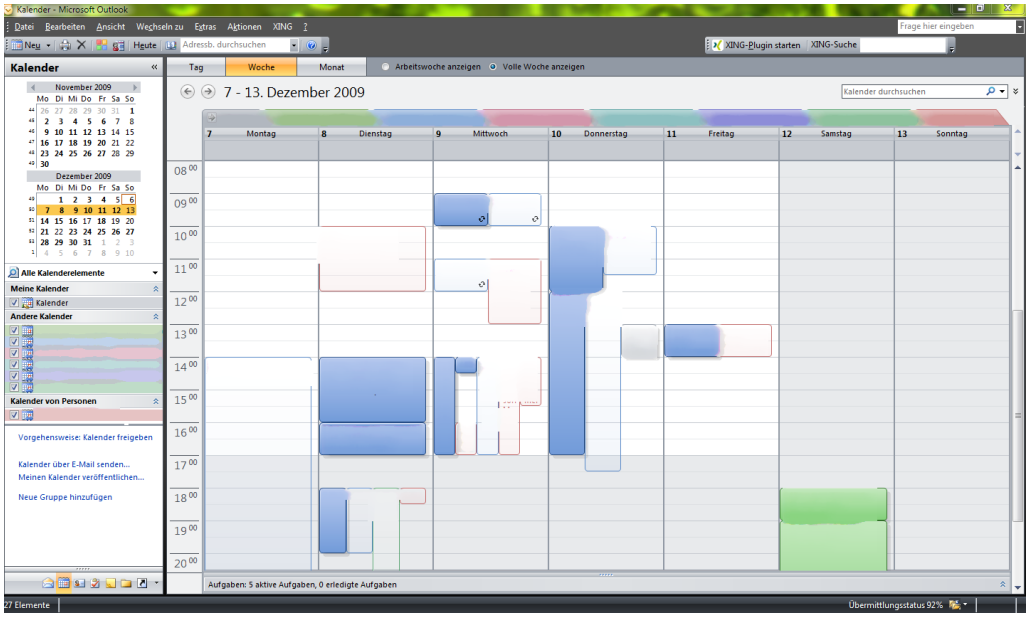


Рисунок 1.1 – Головна сторінка «Sones GraphDB Console»

– «Sones GraphDB API»: Набір інтерфейсів програмування для взаємодії з базою даних з різних мов програмування, включаючи Java та .NET;

– інтеграція з іншими інструментами: Sones GraphDB може інтегруватися з іншими інструментами та фреймворками для використання у різноманітних додатках та середовищах. Для цього використовується відкритий код з відкритого репозиторію на GitHub (рис. 1.2).

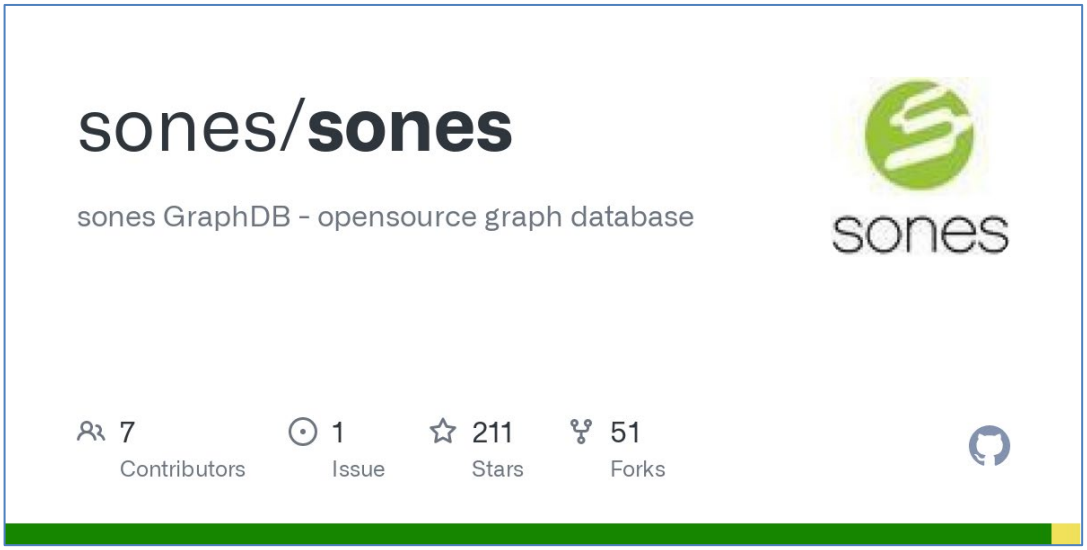


Рисунок 1.2 – Посилання на репозиторій «Sones GraphDB API»

Особливості Sones GraphDB роблять її потужним інструментом для обробки та аналізу графових структур даних в різноманітних сценаріях застосування.

1.5.2 Функціональні можливості СУБД «Neo4J GraphDB»

Neo4j - це графова база даних, розроблена компанією Neo4j, Inc., заснованою Емілем Ейфремом та Яном Ріцмайєром у 2007 році. Основною характеристикою Neo4j є використання графової моделі даних для представлення взаємозв'язків між об'єктами. Для роботи з Neo4j використовується мова запитів Cypher, оптимізована для виразного та ефективного виконання операцій над графовими структурами.

Neo4j надає різні інструменти для взаємодії та розробки. Neo4j Browser - графічний інтерфейс для виконання Cypher-запитів та візуалізації графових даних. Neo4j Desktop спрощує розгортання та керування базами даних. Neo4j Aura - керована хмарна служба для хмарного розгортання. Крім того, Neo4j пропонує драйвери для різних мов програмування, таких як Java та .NET.

За роки свого існування Neo4j стала однією з провідних графових баз даних та знайшла застосування в різних галузях, включаючи телекомунікації, фінанси, логістику та інші галузі, де гнучкі та ефективні графові структури є важливим компонентом обробки даних.

Основні характеристики Neo4j:

– графова модель даних: Neo4j використовує графову модель для представлення даних у вигляді вузлів, ребер та властивостей, що дозволяє легко виражати та виконувати операції зі складними взаємозв'язками;

– мова запитів «Cypher»: Neo4j використовує мову запитів «Cypher», яка спеціально розроблена для роботи з графовими структурами, що полегшує виразність та ефективність запитів;

– транзакційна підтримка: Neo4j підтримує транзакції для забезпечення консистентності та цілісності даних;

– висока продуктивність: Графова база даних Neo4j оптимізована для швидкого виконання запитів на глибоко пов'язаних графах.

Інструменти для роботи з Neo4j:

– «Neo4j Browser» (рис. 1.3): Графічний інтерфейс для виконання Cypher-запитів, вивчення структури графа та візуалізації результатів;

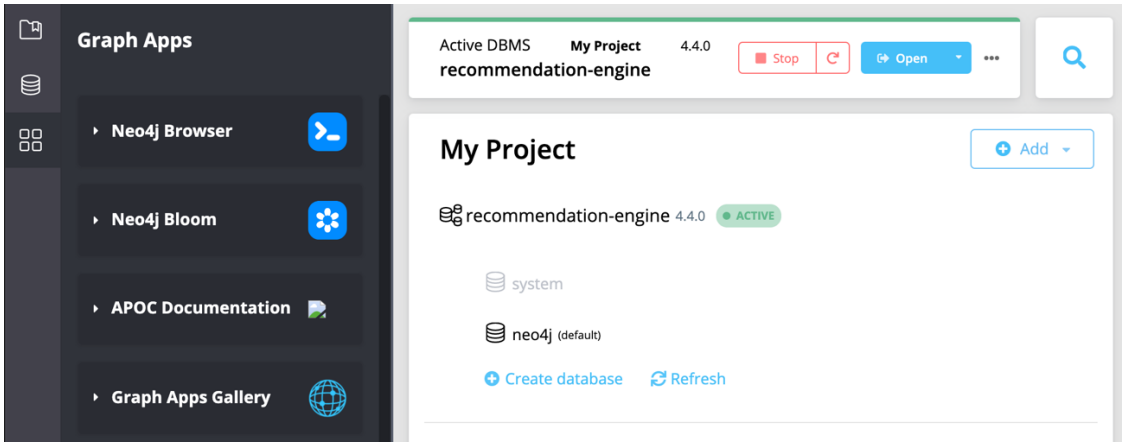


Рисунок 1.3 – інструмент Neo4j Browser

«Neo4j Desktop» (рис 1.4): Зручний інструмент для розгортання та керування локальними та віддаленими інстанціями Neo4j, а також для розробки додатків;

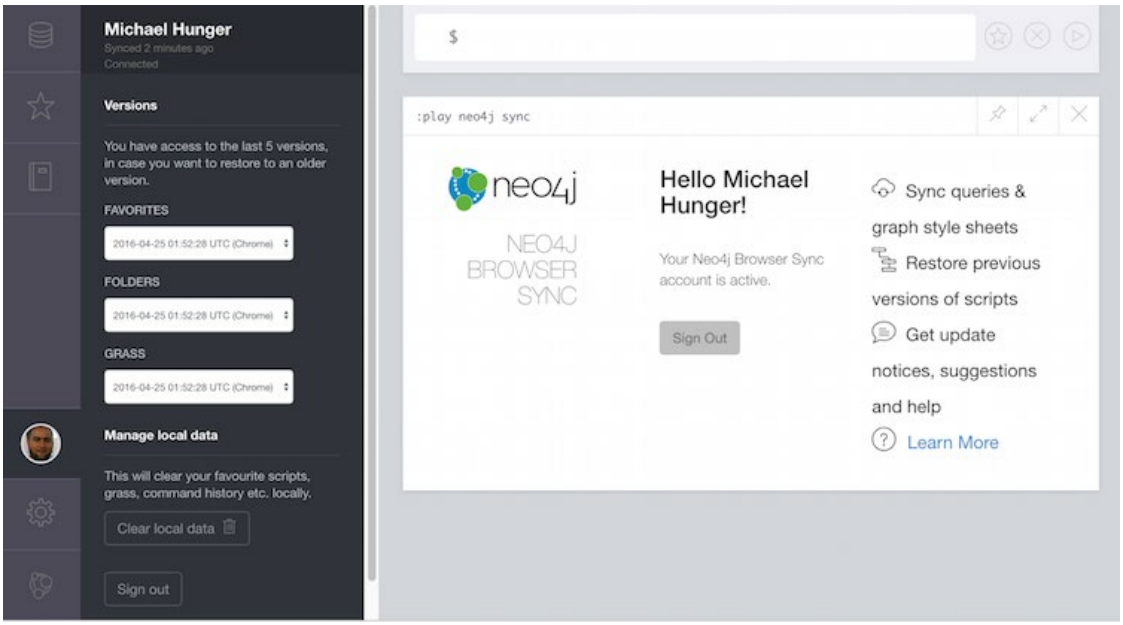


Рисунок 1.4 – інструмент Neo4j Desktop

– «Neo4j Aura»: Керована хмарна служба Neo4j, яка спрощує розгортання та управління базою даних в хмарі;

– «Neo4j Drivers»: Широкий набір офіційних драйверів для різних мов програмування, що дозволяє розробникам легко взаємодіяти з Neo4j у їхньому виборі мови;

– «Neo4j GraphQL» (рис 1.5): Підтримка для GraphQL, яка дозволяє використовувати графові дані у сучасних веб-додатках.

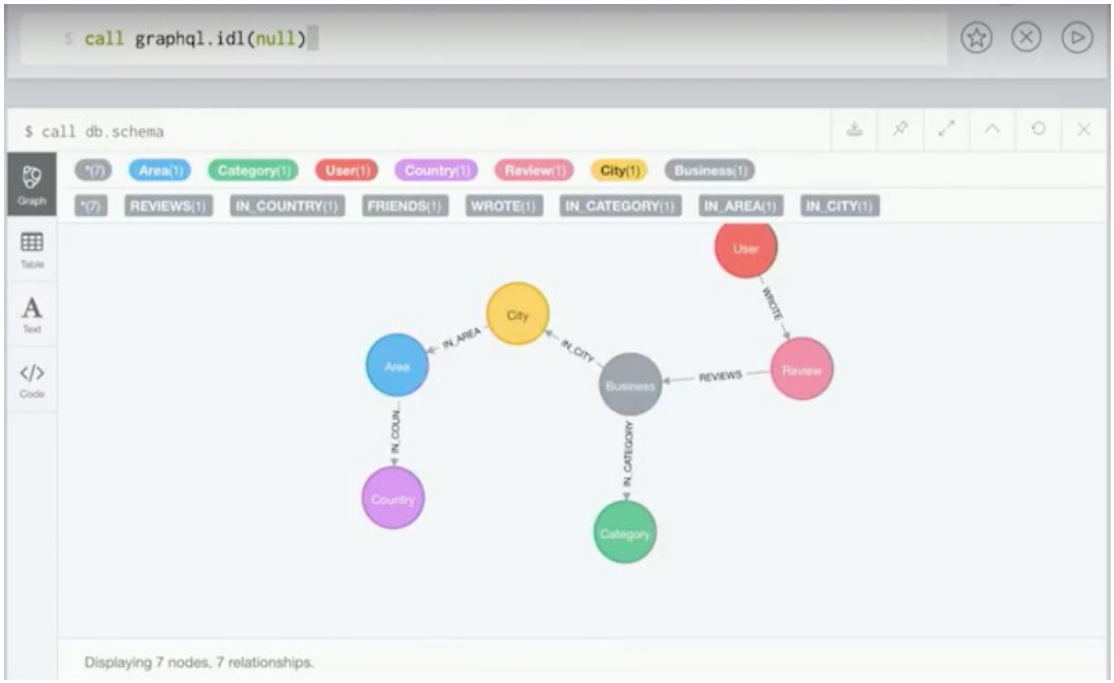


Рисунок 1.5 – інструмент Neo4j GraphQL

Як було зазначено, дуже важливою властивістю саме цієї СУБД є власна декларативна мова запитів «Cypher». Вона дозволяє виразно виконувати операції над графовими структурами та легко взаємодіяти з ними. Ключові слова, такі як «MATCH» для пошуку патернів, «RETURN» для вибору результатів та «WHERE» для фільтрації, надають зручний інтерфейс для роботи з графовими данними. Наприклад, використання синтаксису «CREATE» дозволяє створювати нові вузли з властивостями. Така мова запитів виявляється потужним інструментом для виразного взаємодії з графовими базами даних, надаючи можливості для складних операцій та аналізу залежностей у графі.

Neo4j, одна з провідних графових баз даних, виникла у 2007 році завдяки ініціативі компанії Neo4j, Inc., яку заснували Еміль Ейфрем та Ян Ріцмайєр. Знаходячись у Сан-Матео, Каліфорнія, США, Neo4j вже довгий час займає перше місце серед графових баз даних і широко використовується у сферах, таких як телекомунікації, фінанси, логістика та інші галузі. Ця платформа не лише успішно використовується в різноманітних галузях, але й продовжує розвиватися

та вдосконалювати свої можливості для задоволення вимог сучасних інформаційних потреб.

1.5.3 Функціональні можливості СУБД «DEX»

Система управління графовими базами даних DEX відрізняється рядом характеристик, які підкреслюють його важливість та унікальність.

DEX використовує графову модель даних, де інформація представлена у вигляді вузлів та ребер, сприяючи ефективному вираженню взаємозв'язків між об'єктами. Вона оптимізована для обробки великого обсягу взаємозалежних даних, що забезпечує швидкий доступ до структури графа.

Основні аспекти DEX включають в себе потужну мову запитів, яка дозволяє витягувати дані з графової структури, а також інструменти для зручної роботи з графовими даними. До них входять візуальні інтерфейси для інтуїтивного взаємодії, API для розробників для інтеграції та розширення функціональності, а також інструменти адміністрування для зручного управління та моніторингу.

DEX розроблено з урахуванням специфічних потреб графових даних та надає надійні засоби для взаємодії та оптимізації роботи з цією формою представлення інформації.

1.6 Призначення та область застосування

Сфера використання даної роботи – призначена для застосування у широкому спектрі сучасних індустрій, де графові бази даних демонструють переваги у порівнянні з реляційними. Це може охоплювати області фінансів, телекомунікацій, логістики та багато інших галузей.

Область застосування:

соціальні мережі: графові бази даних використовуються для зберігання та аналізу соціальних зв'язків та мереж великих об'ємів даних;

логістика та транспорт: Управління транспортними маршрутами, слідкування за логістикою та оптимізація руху вантажів;

біографічні дослідження та медицина: Вивчення залежностей в генетичних та біомедичних дослідженнях;

рекомендаційні системи: Побудова персоналізованих рекомендацій для користувачів на основі їх взаємодій та інтересів;

кібербезпека: аналіз та виявлення аномалій у мережевих взаємодіях для забезпечення кібербезпеки.

Функціональне призначення розробленого інструментарію полягає у вимірах та порівняння часової ефективності графових баз даних із реляційними. Він дозволяє визначити оптимальні умови та завдання для використання графових підходів.

Експлуатаційне призначення – призначений для підтримки процесу прийняття рішень у виборі баз даних для конкретних завдань. Це надає можливість вирішувати завдання швидше та ефективніше, зокрема у великих системах та проектах, де швидкість доступу до даних грає важливу роль в успіху вирішення завдань.

Висновки до розділу 1

В результаті аналізу сучасних засобів та СУБД для графових баз даних було визначено поширеність та різницю між цими засобами, перелічено варіативність, розглянути різницю в роботі та властивостях тих чи інших СУБД, які створені під певні задачі та націлені на конкретні результати. Зокрема, можна виділити основні критичні показники, які можуть впливати на швидкість роботи запитів до графової бази даних, серед таких – спосіб зберігання та оперування даними в СУБД, а також індексація та додавання різних метаданих до корисних даних, завдяки яких здійснюється оптимізація в роботі СУБД, впливаючи на її швидкість та місце на носію, яке займає сумарна розгорнута база даних після імпорту.

Такі параметри бувають дуже критичними при вирішенні задач в зберіганні та оперуванні даними, а саме виконання запитів до бази даних. Проблема може заключатись в неправильному вибору типу графової СУБД, або реляційної СУБД, і напряму вплинути на оптимальність виконання часову ефективність використання графової бази даних.

Для вирішення цієї задачі у порівнянні оптимальності та ефективності у застосуванні різних типів баз даних (та СУБД) для них необхідно створити програмне забезпечення, яке дозволить порівняти часову ефективність між наведеними СУБД задля отримання висновків та спростування або доведення факту оптимальності використання графових СУБД для пов’язаних між собою даних.

2 ОБҐРУНТУВАННЯ НАПРЯМКУ ДОСЛІДЖЕНЬ ЧАСОВОЇ ЕФЕКТИВНОСТІ ГРАФОВИХ БД

В процесі проведення досліджень засобів і СУБД для графових баз даних, основний акцент буде зроблено на вимірюванні та порівнянні часової ефективності цих систем у порівнянні з реляційними базами даних. Початковий етап дослідження передбачає аналіз та обґрунтування різних властивостей графових баз даних, враховуючи такі ключові параметри, як методи зберігання даних, операції над графами та механізми індексації.

У межах експериментального підходу планується створення спеціалізованого програмного забезпечення для вимірювання та аналізу часових показників різних операцій над графовими даними. Зокрема, планується врахування часу виконання операцій зчитування, запису, пошуку та аналізу даних у різних сценаріях.

Для експериментів будуть використані тестові сценарії, які відображають типові використання графових баз даних в реальних задачах. Це включає в себе обробку соціальних мереж, логістики та транспортних систем, біографічних досліджень, рекомендаційних систем та кібербезпеки.

Крім того, вимірюватимуться показники часової ефективності для різних об'ємів даних та конфігурацій графових баз даних. Результати досліджень покладуть основу для обґрунтування вибору оптимальних графових баз даних для конкретних завдань порівняно з реляційними аналогами, зокрема з урахуванням часової ефективності та швидкодії роботи.

2.1 Поняття часової ефективності

Для того аби стверджувати чи є СУБД ефективною в порівнянні до іншої необхідно скористатися певною порівняльною оцінкою. Так як однією з найважливіших оцінок роботи певного запиту до бази даних є часова ефективність – було обрано саме цю оцінку.

Також важливою складовою є збільшення обсяг даних, які створюються при відтворенні зв’язків між даними у графових базах даних, тож другою оцінкою

було обрано саме розмір імпортованої бази даних, щоб показати доцільність використання графових баз даних при однакових часових показниках. При отриманні великих обсягів додаткових даних (метадані, які пов’язують дані між собою), та однакових часових показниках – можна прийти до виводу недоцільності використання графової СУБД саме для цієї задачі та при цьому розміру вхідних даних.

2.2 Опис процесу переведення графових даних у реляційні

Проведення досліджень щодо часової ефективності графових СУБД та їх порівняння з реляційними СУБД передбачає вирішення значущої проблеми - наявності різноманітних штампів даних у відповідних системах. Для забезпечення єднання та зрозуміння результатів, необхідно провести перетворення графової моделі даних у реляційну, використовуючи метод "Adjacency List" (Список суміжності).

Сам процес перетворення включає в себе детальну обробку структур графових даних, що забезпечує представлення їх у форматі, зрозумілому для реляційних баз даних. Наприклад, застосування методу "Adjacency List" передбачає перетворення відносин між вершинами графа у відповідні записи та таблиці, використовуючи поняття суміжності. Кожен вузол графа та його зв'язки розкладаються на відповідні рядки та колонки в реляційних таблицях.

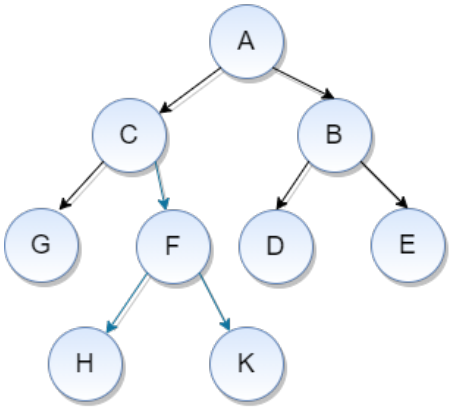
Отримані таким чином дані можна подати у форматі, зрозумілому обом типам СУБД, та забезпечити однакові умови для вимірювання часу виконання запитів. Це дозволяє створити стандартизовану основу для об'єктивного порівняння ефективності різних баз даних.

Далі, в процесі експериментів, буде здійснюватися вимірювання часових показників роботи графових та реляційних СУБД при виконанні різноманітних запитів. Це включає в себе аналіз операцій зчитування, запису, пошуку та обробки даних у визначених умовах тестових сценаріїв. Такий підхід дозволяє отримати об'єктивні результати та забезпечити консистентність при порівнянні часової ефективності графових СУБД та реляційних аналогів. порівняння часової ефективності графових СУБД та реляційних СУБД

відтворюється у різності штампів *даних*, які треба перевести в один формат для порівняння часу виконання запитів до кожної з них. Для цього використовується метод перетворення «Adjacency List» (Список суміжності), який дозволяє перевести графову модель даних у реляційну, після чого можна буде проводити часові експерименти по виконанню запитів до різних типів баз даних.

2.2.1 Основна інформація про метод перетворення Adjacency List

Для цього був використаний так званий Adjacency List (Список суміжності), щоб ми могли перевести графову базу даних у реляційний вид - вид таблиць (рис. 2.1).



TREE		
ID(PK)	NAME	PARENT_ID (FK)
1	A	null
2	B	1
3	D	2
4	E	2
5	C	1
6	F	5
7	H	6
8	K	6
9	G	5

Рисунок 2.1 – Зображення трансформації графової БД у реляційну БД

Щоб описати сутність методу перетворення завдяки списку суміжності, треба описати основний принцип, по якому будується взаємозв'язок між вершинами у вигляді реляційної таблиці. Візьмемо до прикладу неорієнтований циклічний граф (рис. 2.2) і розберемо його взаємодії між вершинами, щоб представити їх у вигляді таблиці.

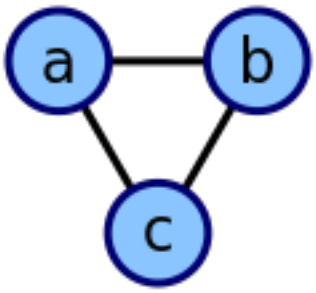


Рисунок 2.2 – Приклад неорієнтованого циклічного графу

У теорії графів та інформатиці використовується підхід, що полягає у представленні наведеного вище графа за допомогою трьох неупорядкованих списків: {b, c}, {a, c}, {a, b}. Ці списки, відомі як список суміжності, представляють собою елементи, які описують зв'язки між вершинами у графі. Кожен з цих невпорядкованих списків розкриває сусідів конкретної вершини, надаючи вичерпну інформацію про структуру графа. У контексті комп'ютерних програм та алгоритмів обробки графів, список суміжності стає одним із часто використовуваних засобів візуалізації та аналізу графових структур даних.

Представлення списку суміжності для графа асоціює кожну вершину в графі з набором її сусідніх вершин або ребер. Існує безліч варіацій цієї основної ідеї, що відрізняються деталями в тому, як вони реалізують зв'язок між вершинами та колекціями, чи вони включають як вершини, так і ребра, чи тільки вершини у ролі об'єктів першого класу, і які типи об'єктів використовуються для представлення вершин та ребер.

Реалізація, запропонована Гвідо ван Россумом, використовує хеш-таблицю для зв'язування кожної вершини в графі з масивом суміжних вершин. У цьому представленні вершина може бути представлена будь-яким хешованим об'єктом. Немає явного представлення ребер як об'єктів.

Кормен Дерінго пропонує реалізацію, в якій вершини представлені номерами індексів. У їхньому підході використовується масив, індексований за номером вершини, де кожна ячейка масиву для кожної вершини вказує на однонапрямлений список сусідніх вершин цієї вершини. У цьому представленні вузли однонапрямленого списку можуть інтерпретуватися як граничні об'єкти; проте вони не зберігають повної інформації про кожне ребро (зберігається лише одна з двох кінцевих точок ребра), а для неорієнтованих графів для кожного ребра існує два різні вузли в списках для кожного з двох об'єктно-орієнтована структура списку інцидентності, запропонована Гудричем і Тамассі, включає спеціальні класи вершинних та краєвих об'єктів. Кожен об'єкт вершини має змінну екземпляра, що вказує на об'єкт колекції, в якому перераховані сусідні граничні об'єкти. У свою чергу, кожен об'єкт краю вказує на два об'єкта вершини у своїх

кінцевих точках. Ця версія списку суміжності використовує більше пам'яті, ніж версія, в якій суміжні вершини перераховані безпосередньо, але наявність явних краєвих об'єктів надає додаткову гнучкість для зберігання додаткової інформації про ребра.

2.2.2 Операції методу перетворення Adjacency List

Основна операція, яку виконує структура даних списку суміжності, полягає у видачі списку сусідів заданої вершини. За допомогою будь-якої з описаних вище реалізацій це може бути здійснено з постійним часом для кожного сусіда. Іншими словами, загальний час для отримання інформації про всіх сусідів вершини v пропорційний ступеню вершини v.

2.2.3 Основні компроміси методу перетворення Adjacency List

Основною альтернативою списку суміжності є матриця суміжності. Це матриця, де рядки та стовпці індексуються вершинами, а кожна комірка містить логічне значення, що вказує, чи існує ребро між вершинами, що відповідають рядку та стовпцю комірки. Для розрідженого графа (де більшість пар вершин не з'єднані ребрами) список суміжності значно більш ефективний за простором, ніж матриця суміжності (зберігається у вигляді двовимірного масиву): використання простору списку суміжності пропорційно кількості ребер і вершин у графі, у той час як для матриці суміжності, збереженої таким чином, простір пропорційний квадрату кількості вершин. Проте матриці суміжності можна ефективно зберігати за простором, відповідно до лінійного використання простору списку суміжності, використовуючи хеш-таблицю, індексовану парами вершин, а не масивом.

Ще однією суттєвою відмінністю між списками суміжності та матрицями суміжності є ефективність виконання ними операцій. У списку суміжності сусіди кожної вершини можуть бути перераховані ефективно за час, пропорційний ступеню вершини. У матриці суміжності ця операція займає час, пропорційний кількості вершин у графі, що може бути значно вище ступеня вершини. З іншого

боку, матриця суміжності дозволяє перевіряти, чи є дві вершини суміжними, за постійний час; список суміжності повільніше виконує цю операцію.

2.2.4 Структура даних для методу перетворення Adjacency List

Для використання у якості структури даних головною альтернативою списку суміжності є матриця суміжності. Оскільки кожен запис у матриці суміжності вимагає лише одного біта, його можна подати дуже компактно, займаючи лише | V | / 8 байтів неперервного простору, де | V | - кількість вершин у графі. Крім зекономленого простору, ця компактність сприяє локалізації посилань.

Проте для розрідженого графа списки суміжності вимагають менше місця, оскільки вони не витрачають марно простір для подання ребер, яких немає. З використанням простої реалізації масиву на 32-бітному комп'ютері, список суміжності для неорієнтованого графа вимагає приблизно 2 · (32/8) | E | = 8 | E | байтів простору, де | E | - кількість ребер у графі.

Зауваживши, що неорієнтований простий граф може мати не більше (| V | - 1) / 2 ≈ V ребер, які допускають петлі, ми можемо позначити d = | E | / | V | як щільність графа. Тоді 8 | E |>| V | / 8, коли | E | / | V |>1/64, тобто представлення списку суміжності займає більше місця, ніж представлення матриці суміжності, коли d>1/64. Отже, граф повинен бути досить розрідженим, щоб виправдати представлення списку суміжності.

Окрім зекономленого місця, різні структури даних також полегшують виконання різних операцій. Знаходження всіх вершин, суміжних з заданою вершиною у списку суміжності, так само просто, як і читання списку. При використанні матриці суміжності, замість того, необхідно просувати всю строку, що займає час O (| V |). Наявність ребра між двома вказаними вершинами можна визначити одразу за допомогою матриці суміжності, при цьому знадобиться час, пропорційний мінімальній ступені двох вершин із списком суміжності.

Також можливо, але менш ефективно, використовувати списки суміжності для перевірки наявності ребра між двома вказаними вершинами. У списку суміжності, де сусіди кожної вершини не впорядковані, перевірка наявності ребра може бути виконана за час, пропорційний мінімальному ступеню двох

заданих вершин, використовуючи послідовний пошук серед сусідів цієї вершини. Якщо сусіди представлені у вигляді впорядкованого масиву, може бути використаний бінарний пошук, час якого пропорційний логарифму ступеню.

Висновки до розділу 2

В даному розділі було визначено основну проблему, яку необхідно було вирішити – неоднаковість структур даних та штампів у графових та реляційних базах даних. Ця проблема стає ключовою при порівнянні часової ефективності графових та реляційних систем у виконанні різноманітних запитів. Для досягнення єднання та об'єктивності у вимірюванні часу виконання, було обрано метод перетворення "Adjacency List," який забезпечує конвертацію графових структур у реляційні та сприяє вирішенню вказаної проблеми.

Важливим аспектом цього перетворення є детальна обробка взаємозв'язків та структур графових даних, щоб забезпечити їх адекватне відображення у форматі реляційних баз даних. Врахування нюансів, таких як різні типи зв'язків, ієрархічність та ваги ребер, дозволяє створити зрозуміле та стандартизоване представлення графових даних в реляційному середовищі.

Такий підхід дозволяє максимально приблизити умови використання графових та реляційних баз даних, створюючи єдиний стандарт для порівняння їх часової ефективності. Впровадження методу "Adjacency List" становить ключовий крок у розробці програмного забезпечення для дослідження та порівняння ефективності цих двох типів систем у реальних умовах, що дозволить отримати об'єктивні висновки та спростувати або підтвердити факт оптимальності графових баз даних у визначених умовах в порівнянні із реляційними аналогами.

3 ПРОЕКТУВАННЯ І РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ

3.1 Зовнішнє проектування

3.1.1 Вхідні дані

Вхідними даним даної системи є:

– імпортовані дані, на яких буде виконуватись експеримент (обрано взаємозв’язки вулиць США та Канади;

– необхідні параметри для авторизації в системі СУБД;

– тип запиту, часова метрика якого буде вимірюватись;

– граничний розмір даних, який повинен бути використаний в конкретному експерименті;

– рівень пошуку сусідніх вершин;

– тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД.

Вхідні параметри мають пройти валідацію.

3.1.2 Вихідні дані

Вихідними даними програмного додатку є файл та проміжні повідомлення про роботу певного експерименту.

Вміст файлу еквівалентний повідомленням які отримує користувач при роботі з користувацьким інтерфейсом. Повідомлення містять наступну інформацію:

– загальний час витрачений на роботу імпорту бази даних;

– час витрачений на запит пошуку сусідніх вершин до заданого рівня;

– час витрачений на запит пошуку перетину множини сусідів;

– розмір імпортованої бази даних.

3.1.3 Формалізація задачі

Для формалізація задачі зовнішнього проектування представлена у вигляді діаграми варіантів використання (рис. 3.1).

Користувач в даному випадку – дослідник, який взаємодіє із програмним забезпеченням представлений у вигляді актора. Прецеденти відображають можливі способи використання даного ПО.

3.2 Базова архітектура системи

Розроблений інструментарій, спрямований на дослідження часової ефективності графових баз даних, володіє простою та зрозумілою структурою, а його користувацький інтерфейс відзначається лаконічністю. Особлива увага приділена можливостям легкого підключення різноманітних графових баз даних та відповідної підтримки їх контрактів через API. Важливо відзначити, що проектування інтерфейсу передбачає його гнучкість, враховуючи можливі майбутні зміни та розширення, включаючи використання в якості окремої бібліотеки. Ця гнучкість досягається за допомогою використання інтерфейсів для основних компонентів системи.

3.3 Внутрішнє проектування

3.3.1 Огляд обраної мови програмування

Вибір мови програмування був серед популярних об’єктно-орієнтованих.

Для створення даного програмного інструментарію було використано мову програмування Java, що базується на платформі Java SE (Java Standard Edition). Java є мовою програмування загального використання, відомою своєю переносимістю та широким застосуванням у різних областях розробки програмного забезпечення.

Платформа Java SE є відкритою та кросплатформеною, що дозволяє виконувати програми на різних операційних системах без необхідності перекомпіляції. Ця особливість відкриває можливість проводження тестувань та експериментів на різних платформах та операційних системах.

Java користується значною популярністю, існує велика спільнота розробників, а також обширний ресурс із документацією та навчальними матеріалами. Розробка під Java можлива у середовищі розробки, такому як IntelliJ IDEA або Eclipse.

Мова Java має кілька переваг, серед яких:

– система строгої типізації даних, що допомагає виявляти помилки на етапах розробки та компіляції;

– об'єктно-орієнтований підхід для швидкого розширення систем;

– система автоматичного управління пам'яттю (Garbage Collection), яка полегшує вирішення проблем витоку пам'яті;

– велика кількість вбудованих бібліотек для різних завдань;

– підтримка асинхронного виконання методів;

– використання технології LambdaJ (Java 8 і вище) для роботи з колекціями та потоками даних;

– можливість створення багатопоточних програм.

3.3.2 Можливості дослідження часу

Для виконання операцій, пов'язаних із вимірами виконаного часу в межах певного коду, використовуються класи мови програмування Java, такі як System.currentTimeMillis() та System.nanoTime(). Ці класи надають можливість виконання точних вимірів витраченого часу [47, 48].

Для виконання вимірів часу протягом певного інтервалу необхідно використати вбудовані методи. Для початку вимірювання часу використовується System.nanoTime() (startTime), а для зупинки – System.nanoTime() (endTime). Для отримання значення часу, витраченого на виконання, використовується різниця між endTime та startTime.

Щоб отримати інформацію про точність та розподільну здатність реалізації часу, можна використати метод System.nanoTime(). Цей метод повертає значення, що представляє час в наносекундах.

Для отримання значення загального витраченого часу можна використовувати різні одиниці виміру, такі як мілісекунди, секунди чи такти таймеру:

```
long elapsedNanoSeconds = endTime - startTime;
long elapsedMilliseconds = TimeUnit.NANOSECONDS.toMillis(elapsedNanoSeconds);
long elapsedSeconds = TimeUnit.NANOSECONDS.toSeconds(elapsedNanoSeconds);
```

Отже, для виконання вимірів витраченого часу на певному проміжку використовується вбудований функціонал мови Java для роботи з часом.

3.3.3 Зменшення впливу кешу

Так як заміри часу роботи алгоритму проводяться багаторазово для одних і тих конфігурацій необхідно враховувати вплив кешу центрального процесору.

Пам’ять кешу – це високошвидкісна пам’ять, яка є частиною центрального процесору [9]. Дана пам’ять надає можливість пришвидшити доступ до оперативної пам’яті шляхом збереження даних, які часто використовуються, в пам’яті кешу.

Багаторазове виконання одного і того самого функціоналу може призвести до такого випадку, коли часові показники отримані під час проведення перших виконань будуть нижчими ніж часові показники подальших виконань.

3.3.4 Зменшення впливу кешу

Кеш-пам'ять в унікальний спосіб вписується в архітектуру центрального процесора [9]. Ця високошвидкісна форма пам'яті розташована безпосередньо в процесорі і використовується для прискорення доступу до оперативної пам'яті. Суть її функціонування полягає в зберіганні інформації, яка використовується часто, в спеціальному буфері, що дозволяє значно зменшити час доступу до цих даних.

Використання кеш-пам'яті може бути у порівнянні зі спеціальним сховищем, яке швидко надає необхідні ресурси для оптимальної продуктивності центрального процесора. Відмінною рисою є можливість забезпечити тимчасове збереження інформації, що дозволяє уникнути знову повторюваних операцій зовнішнього доступу до пам'яті.

Завдяки цьому механізму кеш-пам'яті стає важливим елементом для оптимізації швидкодії центрального процесора, забезпечуючи ефективну роботу з даними та виключаючи зайві затримки у випадках частого використання конкретних ресурсів.

Багаторазове виконання одного і того самого функціоналу може призвести до ситуації, коли часові показники, отримані під час проведення перших виконань, будуть нижчими, ніж часові показники подальших виконань.

Для зменшення впливу кешу на результати експериментів планується використовувати віртуальні машини. Для кожного проведеного дослідження буде створюватись новий екземпляр віртуальної машини, а також новий експорт даних у базу даних [10]. Це дозволить зменшити можливий вплив кешу, оскільки кожен експеримент буде виконуватися у власному ізольованому середовищі. Зазначимо, що для експериментів з однією базою даних можливо проводити дослідження без видалення віртуальної машини, забезпечуючи при цьому надійність та послідовність проведення експериментів.

3.3.5 Система логування

Під Під час розробки та вдосконалення програмного забезпечення використання ефективних інструментів логування в Java є ключовим аспектом. Для цієї мови програмування обрано логер Log4j [13], що є потужною і безкоштовною бібліотекою логування, сумісною з багатьма платформами Java.

Log4j дозволяє одночасно записувати необхідні повідомлення до різних ресурсів, таких як бази даних, файли та консоль. Його гнучкість полягає в тому, що розробник може конфігурувати параметри логування не лише перед запуском програми, а й протягом її виконання.

За допомогою системи шаблонів у конфігураційному файлі розробник може визначити порядок інформації, яка має бути записана, додатково до цільового повідомлення. Ця інформація може включати дату, час, ім'я класу, який надсилає повідомлення, інформацію про винятки у випадку їх виникнення та рівень логування (debug, info, trace, error, fatal).

Оскільки операції вводу-виводу можуть бути затратними з точки зору продуктивності, Log4j надає можливість використовувати асинхронний механізм через відповідні налаштування у конфігураційному файлі log4j.xml. Це зменшує вплив операцій виводу та сприяє покращенню продуктивності.

Також для ефективного контролю обсягу логування та забезпечення оптимальної продуктивності у випадках, коли потрібно відображати проміжні результати, використання рівневих обмежень логера в Java є розумним підходом.

3.3.6 Взаємодії класів системи

Під час створення архітектури системи було отримано наступну діаграму класів та інтерфейсів (рис. 3.2). Слід урахувати що на діаграмі зображено абстрактний шар загальної системи необхідної для її повноцінного функціонування.

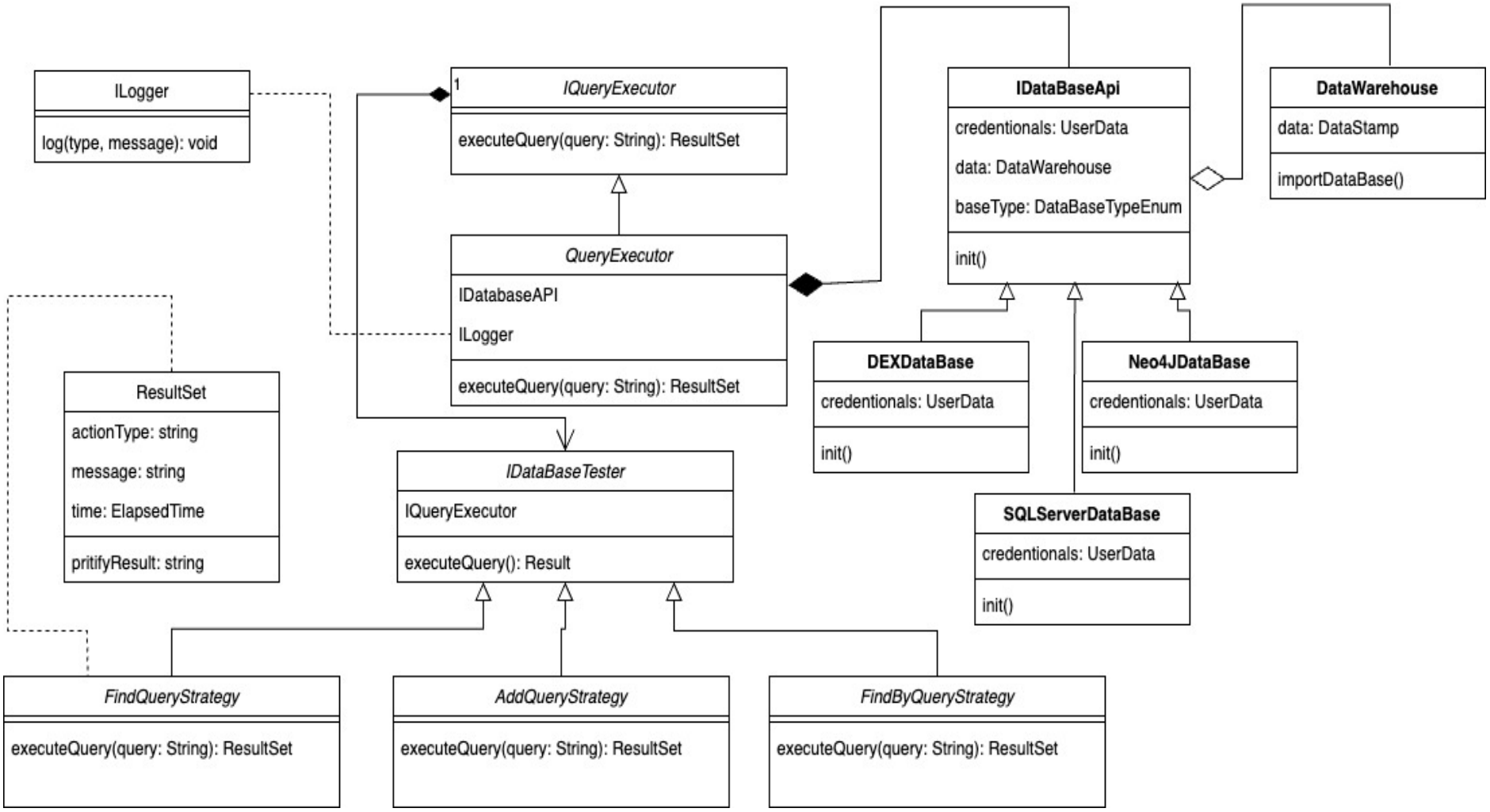


Рисунок 3.2 – UML схема взаємодії основних елементів системи

За рахунок використання поліморфізму було отримано наступні ієрархічні зв’язки:

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні реалізовувати інтерфейс IDatabaseApi;

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні успадковувати клас BaseEntryDatabase;

– усі варіанти логування повинні реалізовувати інтерфейс ILogger;

– інтерфейс IDataBaseTaster повинен використовувати тільки реалізації інтерфейсу IDatabaseApi для відтворення запитів до бази даних (рис 3.5).

Ці відносини забезпечують стабільність та можливість розширення системи, дозволяючи легко додавати нові компоненти чи змінювати існуючі, при цьому забезпечуючи відповідність до визначених інтерфейсів.

Таблиця 3.1 – Основні класи та інтерфейси

Назва елементу	Опис
DatabaseTester	Відповідає за тестування часової ефективності графових та реляційних баз даних.
IDatabaseApi	Інтерфейс який дає контракт використання будь якої бази даних. Послідовники GraphDatabase та RelationalDatabase визначають типи баз даних, з якими система взаємодіє. GraphDatabase відповідає за графові бази даних, тоді як RelationalDatabase представляє реляційні бази. Ці класи надають необхідний інтерфейс для виконання запитів та управління даними відповідного типу.
QueryExecutor	Відіграє роль виконавця запитів до баз даних. Він отримує SQL- або графові запити від інших частин системи та відповідає за їхнє виконання, повертаючи результати у вигляді ResultSet.

Продовження таблиці 3.1

Назва елементу	Опис
ResultSet	Служить для представлення результатів виконаних запитів до бази даних. Його завдання - забезпечити структуроване представлення даних, отриманих в результаті виконання запитів.
ILogger	Визначає загальний інтерфейс для логування подій та даних в системі. Цей інтерфейс дозволяє різним класам системи взаємодіяти із системою логування.
FileLogger та ConsoleLogger	Реалізують інтерфейс ILogger і відповідають за логування інформації у файлі та консолі відповідно. Ці класи забезпечують різні методи виведення інформації.
ExperimentResult	Клас, що представляє результати конкретного експерименту. Він містить інформацію про часові показники, статистику та інші деталі, які можуть бути важливими для подальшого аналізу.
DataWarehouse	Клас визначає сховище даних, де зберігаються результати експериментів. Цей клас дозволяє управляти та зберігати експериментальні дані для подальшого використання та аналізу.
DataStamp	Клас відповідає за створення або імпорт штампу даних, який використовується для налаштування експерименту.
Node, Edge, Table, Column, DataType	Визначають основні елементи, що використовуються для представлення структури даних у графових та реляційних базах даних.

Закінчення таблиці 3.1

Назва елементу	Опис
GraphData та RelationalData	Визначають структуру та особливості графових та реляційних даних відповідно. Вони представляють об'єкти, з якими буде взаємодіяти система.
DataBaseTransformator	Класо, який відповідає за трансформацію даних. Він може забезпечити конвертацію даних з одного формату до іншого для оптимізації експерименту.

3.3.7 Принципи та шаблони проектування

Під час проектування внутрішньої структури системи використовувався принцип SOLID [23]. Даний принцип надає можливість створювати системи, які можна буде легко підтримувати і розширювати функціонал у майбутньому. Назва даної методології містить в собі абревіатуру ключових принципів, з яких складається SOLID:

– The Single Responsibility Principle – принцип єдиної відповідальності. Кожен клас повинен мати тільки одну причину для внесення змін;

– The Open Closed Principle – принцип відкритості / закритості. Програмовані сутності мають бути закриті для модифікацій проте відкриті для розширення;

– The Liskov Substitution Principle – принцип підстановки Барбари Лісков. Клас – спадкоємець повинен доповнювати базовий клас, а не змінювати його;

– The Interface Segregation Principle – принцип розділення інтерфейсу. Краще використовувати багато спеціальних інтерфейсів ніж один загальний;

– The Dependency Inversion Principle – принцип інверсії залежностей. Необхідна присутність залежностей на абстракціях, а не на конкретних реалізаціях.

При розробці архітектури програми для визначення часової ефективності графових баз даних можна використовувати ряд принципів та шаблонів проєктування. Нижче наведено деякі з них:

– принцип єдиної відповідальності (Single Responsibility Principle): клас DatabaseTester вже реалізує функціонал тестування баз даних, QueryExecutor - виконання запитів, ILogger - логування. Кожен клас має чітко визначену відповідальність, що забезпечує чистоту та прозорість коду;

– принцип відкритості/закритості (Open/Closed Principle): система побудована з урахуванням можливості розширення новими класами для різних типів баз даних. Код вже закритий для змін, але відкритий для розширень;

– принцип заміщення Лісков (Liskov Substitution Principle): усі класи баз даних вже реалізують інтерфейси GraphDatabase та RelationalDatabase. Об'єкти базового класу можна замінювати його похідними без порушення коректності системи;

– шаблон Стратегія (Strategy): клас DatabaseTester вже містить механізм вибору стратегії тестування, де різні стратегії реалізуються як окремі об'єкти (рис. 3.3);

– шаблон Декоратор (Decorator): застосування декоратора для функціональності логування реалізовано через класи FileLogger, ConsoleLogger та інші, які додають можливість логування до основного класу (рис. 3.4);

– шаблон Фабричний метод (Factory Method): система вже використовує фабричний метод для створення об'єктів різних баз даних без прив'язки до конкретних класів (рис. 3.5);

– шаблон Оголошення процедури (Observer): клас ExperimentResult вже є спостерігачем, який отримує та обробляє повідомлення про результати експериментів від інших частин системи (рис. 3.6);

– шаблон Ланцюг відповідальності (Chain of Responsibility): система вже містить ланцюг об'єктів логування, які обробляють логи та передають їх далі у ланцюгу;

– шаблон Фасад (Facade): застосування фасаду реалізовано для об'єднання складних підсистем, таких як логування, та надання простого інтерфейсу для взаємодії;

– шаблон Знімок (Memento): система вже використовує шаблон "знімок" для збереження та відновлення стану експериментів.

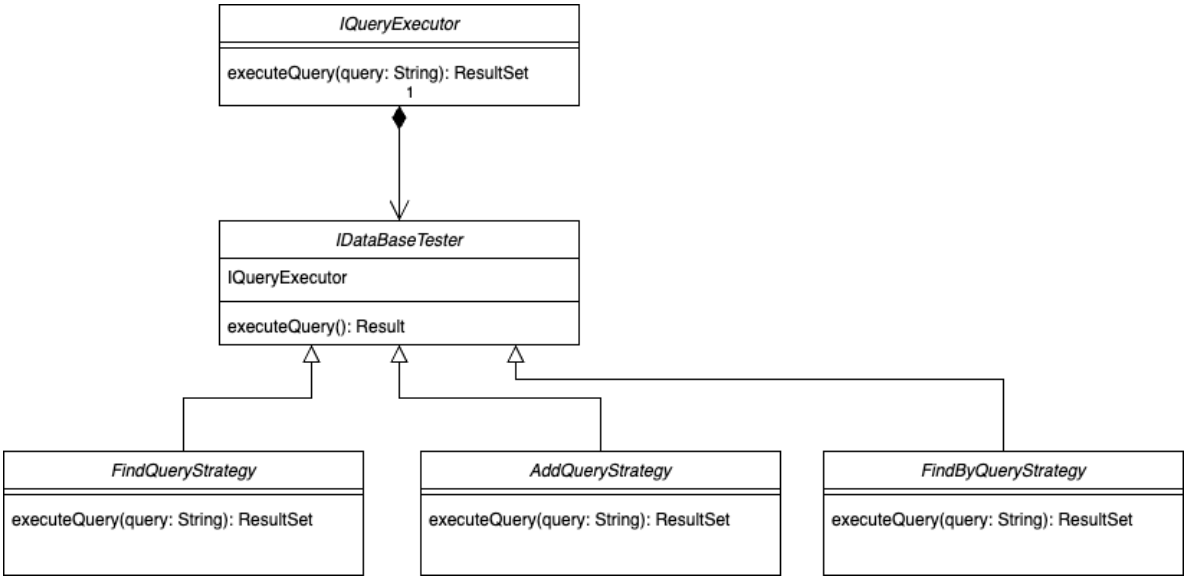


Рисунок 3.3 – Приклад використання спрощеної версії шаблону «абстрактна фабрика»

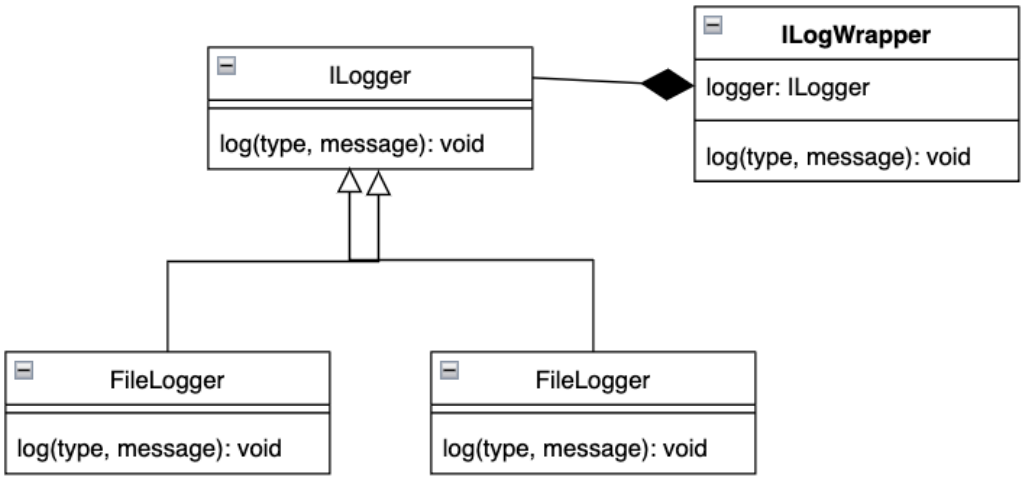


Рисунок 3.4 – Приклад застосування шаблону Decorator

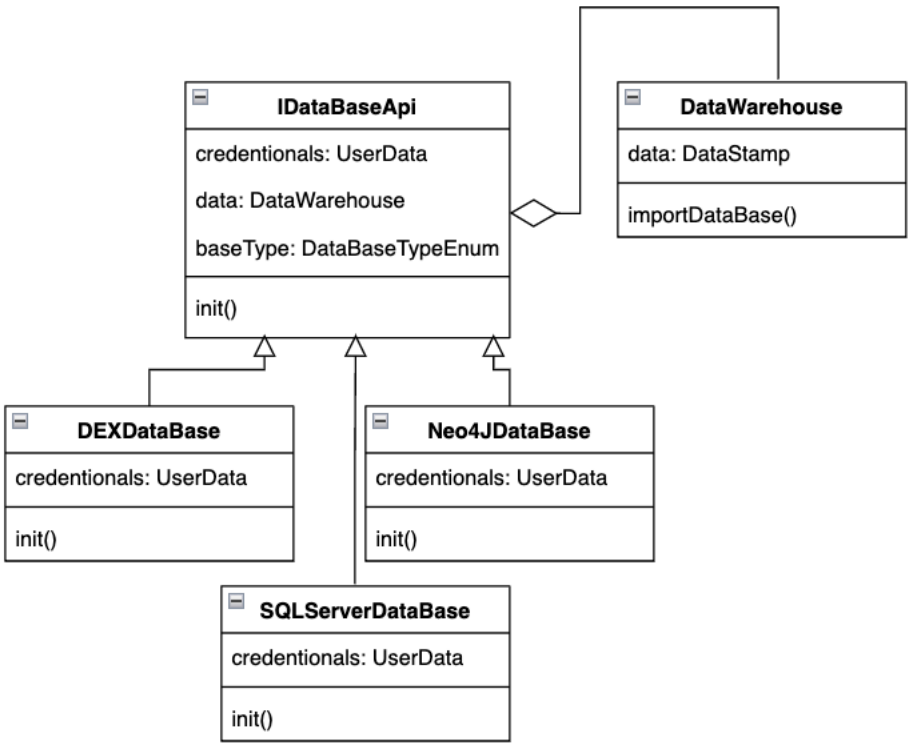


Рисунок 3.5 – Приклад застосування шаблону «Абстрактний метод»

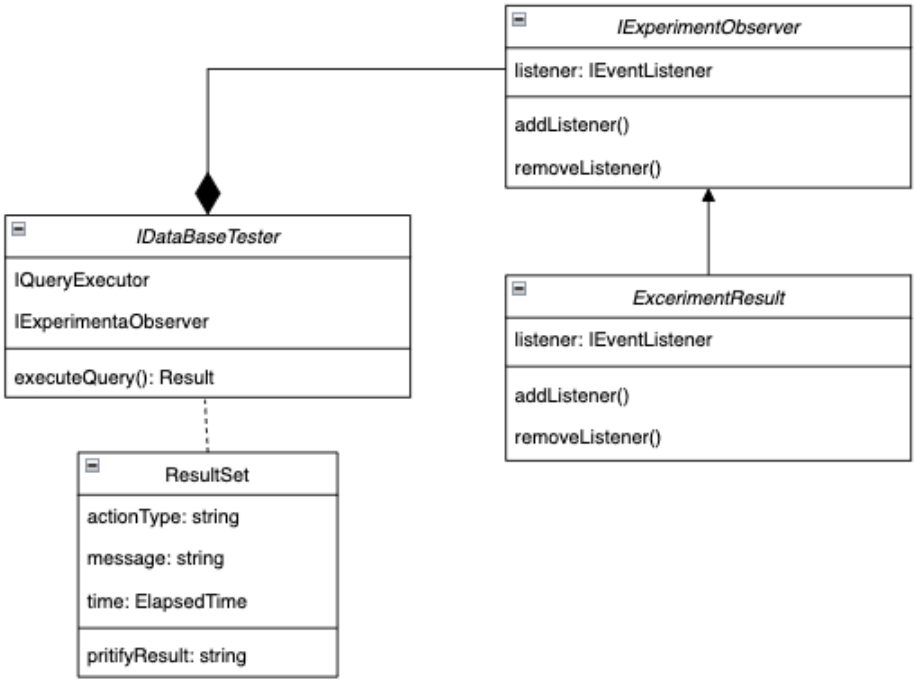


Рисунок 3.6 – Приклад застосування шаблону «Observer»

3.4 Проектування користувацького інтерфейсу дослідника

Під час проектування інтерфейсу необхідно виділити головну ціль створення додатку, оцінивши при цьому мінімальний функціонал необхідний для отримання результатів дослідження.

Інтерфейс користувача був створений у вигляді консольного вікна, що дозволяє взаємодіяти через послідовний вибір або введення необхідних параметрів. За умови мінімальних залежностей між компонентами системи, програмний продукт наділений можливістю легкого розширення та зміни моделі представлення користувацького інтерфейсу за допомогою альтернативної реалізації.

Головні елементи які мають бути заповнені користувачем представлені у таблиці наведеній нижче (табл. 3.2).

Таблиця 3.2 – Головні елементи визначення користувачем

Параметри	Опис
Імпортовані дані, розмір вибірки	Користувач повинен зазначити вже сформовану базу даних для того, щоб використовувати ці дані для подальшого експерименту.
Тип запиту до бази даних	Користувач повинен вибрати який саме тип запиту до бази даних буде досліджуватись.
Рівень пошуку сусідніх вершин	Користувач повинен вибрати рівень пошуку сусідів при типі запиту «пошук

Висновки до розділу 3

У цьому розділі надано опис процесу зовнішнього та внутрішнього проектування інструментального забезпечення, необхідного для здійснення досліджень, зокрема, вимірювання часу, витраченого на виконання запитів до різних баз даних, таких як графові та реляційні, а також порівняння їх властивостей, характеристик та метрик (часу та розміру імпортованої бази даних).

Система, розроблена з використанням шаблонів проектування, володіє можливістю ефективного розширення функціоналу в залежності від потреби, завдяки розподіленню конкретних обов'язків між визначеними класами.

4 ДОСЛІДЖЕННЯ ЧАСОВОЇ ЕФЕКТИВНОСТІ ГРАФОВИХ БАЗ ДАНИХ

4.1 Підготовка до експерименту

Дослідження присвячене визначенню часової ефективності графових баз даних порівняно зі звичайними реляційними базами даних та NoSQL базами даних. Предметом дослідження будуть визначені часові проміжки часу, який потрібен кожній системі управління базами даних (або доданому програмному забезпеченню у вигляді SDK) не те, щоб виконати однаковий перелік операцій по стандартній роботі з сучасними базами даних.

Для вирішення цієї задачі проведено низку експериментів по будуванню схожої бази даних з однаковими взаєминами між об’єктами у даній базі даних. У рамках цих експериментів було виконано:

– побудування бази даних с різним об’ємом та наповненістю (реляційної та графової);
– заміри часу імпорту даних;
– заміри часу пошуку сусідньої вершини до максимального рівня;
– заміри часу пошуку перетину множини сусідів для двох вершин;
– порівняння розміру імпортованої бази даних.

Для проведення експерименту було використано 3 різновиди графових баз даних та одну реляційну базу даних:

– Sones GraphDB (графова база даних);
– Neo4J GraphDB (графова база даних);
– DEX GraphDB (графова база даних);
– Microsoft SQL Server 2012 (реляційна база даних).

Під час експерименту використовувалися останні версії БД, доступні на даний момент: Sones v.2.1, Neo4J v.1.9 та DEX v.4.8. Усі тестові операції реалізовані мовою програмування C# (Sones, DEX) і Java (Neo4J) з допомогою API, наданого кожної з БД. Для конфігурування застосовувалися стандартні налаштування, а також деякі рекомендації з офіційної технічної документації: використання

BatchInserter при імпорті даних в Neo4J і цілих ідентифікаторів в DEX. У Microsoft SQL Server 2012 створюються дві таблиці (для списку вершин і для списку ребер), а обхід графа реалізований за допомогою процедури, що зберігається.

Звісно, реляційні бази даних та графові мають різній варіант зберігання даних, тож є спеціальна процедура переносу графових вершин та ребер у реляційний вигляд. Це потрібно для того, щоб бази даних були максимально схожі, так як операції, які ми будемо досліджувати потрібно робити на однакових штампах даних для максимально оптимальних результатів.

4.2 Опис процесу переведення графових даних у реляційні

Для виконання цього завдання використовувався метод, відомий як "Список суміжності" (Adjacency List), щоб перетворити графову базу даних у реляційний формат, представлений у вигляді таблиць.

Описуючи сутність методу перетворення за допомогою списку суміжності, необхідно пояснити основний принцип, за яким встановлюється зв'язок між вершинами у вигляді реляційної таблиці. У теорії графів і інформатиці використовується підхід, що полягає у представленні графа за допомогою трьох неупорядкованих списків: {b, c}, {a, c}, {a, b}. Ці списки, відомі як "список суміжності", містять елементи, які вказують на зв'язки між вершинами у графі. Кожен з цих списків розглядає сусідів конкретної вершини, надаючи повну інформацію про структуру графа. У світі комп'ютерних програм і алгоритмів обробки графів, список суміжності широко використовується для візуалізації та аналізу графових структур даних.

Представлення списку суміжності для графа асоціює кожну вершину графа з набором її сусідніх вершин або ребер. Існує безліч варіацій цієї основної ідеї, відзначаючись деталями у тому, як вони реалізують зв'язок між вершинами та колекціями, чи вони включають як вершини, так і ребра, чи тільки вершини у ролі об'єктів першого класу, і які типи об'єктів використовуються для представлення вершин та ребер.

4.3 Підготовка реальному штампу даних BigData

Big Data – це концепція, яка описує великі обсяги даних, які вирізняються не лише своєю об'ємною, але іншими важливими характеристиками. Основні аспекти великих даних включають великий обсяг (Volume), велику швидкість (Velocity), різноманітність (Variety), правдоподібність (Veracity) та цінність (Value).

Об'єм даних може бути величезним і походити з різних джерел, таких як соціальні мережі, мобільні додатки, датчики, сенсори, логи, відео та багато інших. Наприклад, дані від сучасних смартфонів, які постійно надходять з численних джерел, можуть бути величезними за обсягом.

Швидкість збору та обробки даних – ще один ключовий аспект великих даних. Наприклад, транзакції в онлайн-магазинах, поштові дані або дані від датчиків машин можуть генерувати велику кількість інформації в режимі реального часу.

Різноманітність даних вказує на їхню різноманітність та формати. Це може бути текст, зображення, відео, аудіо, графи чи структуровані таблиці. Наприклад, дані з соціальних мереж часто включають текст, зображення та відео.

Важливість правдоподібності полягає в тому, щоб можна було вірити в якість та достовірність даних. Це особливо важливо, коли мова йде про прийняття бізнес-рішень або проведення досліджень.

Цінність даних полягає в тому, що вони повинні приносити користь та використовуватися для прийняття обґрунтованих рішень.

Для нашого дослідження часової ефективності графових та реляційних баз даних був використаний штамп даних. Це поєднання великої кількості вулиць в Америці (рис. 4.1) та Європи (рис. 4.2), зібраних у формі бази даних, яке репрезентує різноманітність і об'єм великих даних. Наприклад, на момент 24 жовтня 2021 року OpenAddresses обробило 578,632,464 адреси з 2,601 джерел, підкреслюючи великий обсяг та різноманітність цих даних.

Для візуальної репрезентації даних, що характеризують вибірку адрес, нижче подано графічні моделі, які відтворюють дані, що використовувалися для створення імпортованої бази даних, необхідної для проведення експерименту.

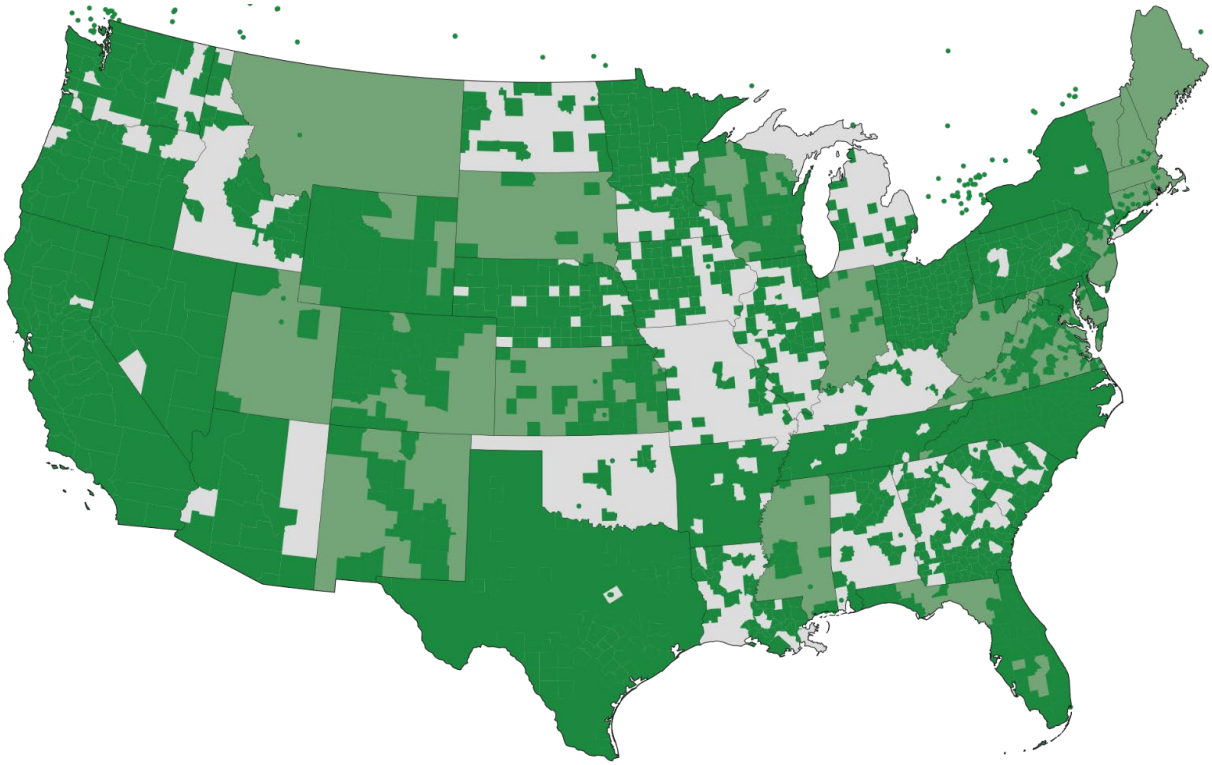


Рисунок 4.1 – Графічне відображення вибірки даних: адреси Америки

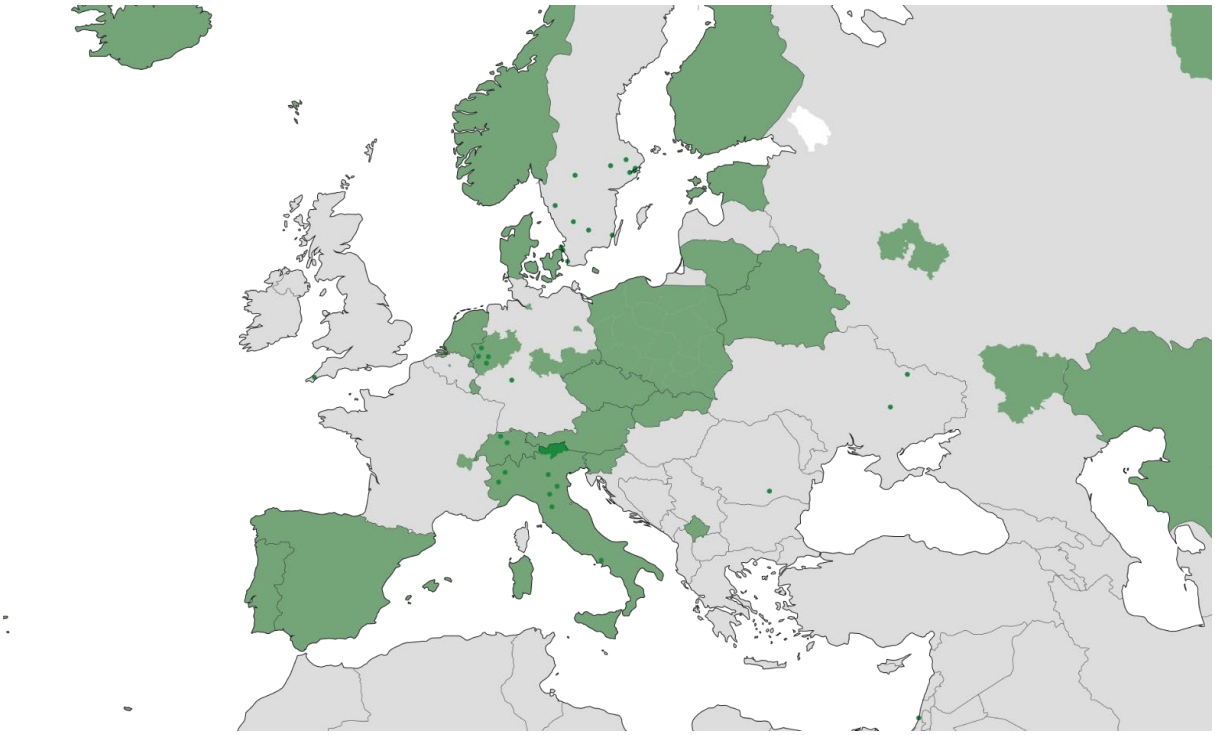


Рисунок 4.2 – Графічне відображення вибірки даних: адреси Європи

4.4 Проведення експерименту

Для дослідження пошуку були використані 2 варіанти операцій пошуку:

пошук всіх сусідів, що знаходяться на певному віддаленні від заданої вершини. Для цього була реалізована функція, яка приймає параметр n, а повернутися повинні всі сусідні вершини, до яких мінімальна відстань не більше n;

пошук всіх перетинів знайдених сусідів двох вершин на певному віддалені від заданих вершини (через параметр n).

Експерименти виконувались у тому порядку, як вони представлені вище: імпорт, пошук сусідів, пошук перетину множин сусідів. Якщо виконання не завершувалося протягом 12 годин, операція переривалася у зв'язку з неактуальністю такого тривалого виконання на запропонованому обсязі даних. Вимірювання кожної з баз даних на імпорт, 2ох варіантів пошуку та розмір отриманої бази – проводилось 4 рази поспіль, щоб вилучити вплив на часові показники пошуку та імпорту даних в розрізі кеша оперативної пам’яті, кешу роботи попереднього алгоритму, або інших програмних оптимізацій, які могли вплинути на час роботи СУБД.

Також, після виконання пошуку по тому самому запиту в базу даних, ми маємо оптимізацію у вигляді кеш файлів самої СУБД, які роблять додаткову індексацію файлів у певних хеш-таблицях, тобто потрібно зробити варіації пошуку без імпорту нової бази, щоб порівняти варіативність оптимізаційних алгоритмів кожної СУБД вже на «розігрітому середовищі».

Тестовий стенд, на якому проводилося тестування має наступну конфігурацію:

- Intel Core i5 3.1 GHz, 8 Gb DDR3, 1 Tb 7200 rpm hard drive;
- Windows Server 2008 x64. У БД DEX та Neo4J значення розміру кешу;
- встановлювалося рівним 4 Gb.

Тестовий стенд, використаний для проведення експериментів, володіє конфігурацією, спеціально адаптованою для вимірювання та оцінки

продуктивності графових баз даних. Основна пристосованість полягає в характеристиках обладнання та використанні віртуальних машин, що забезпечують необхідні умови для проведення досліджень.

Віртуальна машина (ВМ) є віртуалізованим емулятором апаратного забезпечення, який дозволяє використовувати віртуальне середовище на фізичному комп'ютері. В даному випадку, використання віртуальних машин має декілька обґрунтованих переваг для проведення досліджень ефективності графових баз даних.

Використання віртуальних машин надає уніфіковані умови для порівняння різних СУБД. Це досягається завдяки створенню ізольованого середовища для кожної тестирувальної системи, що дозволяє уникнути впливу внутрішніх налаштувань та конфігурацій, що може відбуватися в реальних умовах.

Однією з важливих переваг використання віртуальних машин є можливість конфігурування параметрів процесора та пам'яті для емуляції різних обставин. Вказані параметри включають такі як обсяг доступної пам'яті (RAM), об'єм жорсткого диска, швидкість процесора та інші. Це робить можливим створення умов, аналогічних реальним, але при цьому контрольованих та відтворюваних.

Окрім того, параметри запуску віртуальних машин для конкретних СУБД, таких як Neo4J, можуть бути точно налаштовані, включаючи величину кешу та параметри Java Virtual Machine. Наприклад, використання параметрів -d64 -server -Xmx4096m -XX:+UseConcMarkSweepGC при запуску Java Virtual Machine для Neo4J визначається деталізованою конфігурацією, що дозволяє максимально оптимізувати продуктивність.

Виртуальні машини також ізолюють експериментальне середовище від інших процесів та програм, що виконуються на основній операційній системі. Це дозволяє уникнути непередбачених впливів і забезпечує більш об'єктивні результати експериментів.

Таким чином, використання віртуальних машин у дослідженнях продуктивності графових баз даних є важливою технікою, що забезпечує контрольовані, однак об'єктивні умови для порівнянь та оцінок різних СУБД.

Для тестування було запропоновано чотири набори даних, що містять різну кількість вершин та ребер, структура графа. У найменшій кількості об'єктів приблизно дорівнює 1 млн, а найбільше – 100 млн. Результати дослідження часової ефективності кожної бази даних наведені у таблицях 4.1-4.4

Таблиця 4.1 – Результати досліджень при 16 000 вершин та 600 000 ребер

Назва БД	Час (с)				Розмір отриманої БД (після імпорту), Мб
	№ експерименту	Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
Sones	1	5	11	15.9	19
	2	4	11.2	17.1	19
	3	4	10.9	16.3	19
	4	4	11.3	15.9	19
	5	4	11.4	15.9	19
	6	4	11.3	15.8	19
	7	4	11.2	15.9	19
	8	4	11.3	16.2	19
Neo4J	1	6	3	9	21
	2	5	3	9	21
	3	6	3.1	9	21
	4	5	2.9	9	21
	5	5	2.9	9	21
	6	5	3.2	9	21
	7	6	3.1	9	21
	8	5	3.1	9	21

Закінчення таблиці 4.1

Назва БД	Час (с)				Розмір отриманої БД (після імпорту), Мб
	№ експерименту	Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
DEX	1	4	8	24	22
	2	4	7	27	22
	3	4	8	24	22
	4	3.8	8	23.5	22
	5	4	7	23.7	22
	6	3.9	7	24	22
	7	3.8	7	24.1	22
	8	3.9	7	24	22
SQL Server	1	9	5.1	20	45
	2	8	3.8	18.3	45
	3	8.5	4.1	18.3	45
	4	8	3.9	18.6	45
	5	8	3.9	18.2	45
	6	8	3.7	18.3	45
	7	8.2	3.7	18.4	45
	8	8.1	3.8	18.4	45

З виведених результатів експерименту, де розглядалися СУБД Sones, DEX та Neo4J на обсягах даних, складаючих 16 000 вершин та 600 000 ребер, виявлено відсутність переваг у швидкості виконання запитів та імпорту штампу бази даних порівняно з реляційною СУБД MS SQL Server. Однак варто відзначити, що графова СУБД Neo4J відзначається подвійно швидшим імпортом, на 50% ефективнішим виконанням запитів пошуку сусідів та більш ніж вдвічі

покращеною швидкістю пошуку перетину множини сусідів. Таким чином, в даному експерименті лише Neo4J продемонструвала результати, що перевершують реляційну СУБД, в той час як інші показали подібні або менш ефективні характеристики.

Таблиця 4.2 – Результати досліджень при 100 000 вершин та 1 200 000 ребер

Назва БД	Час (с)				Розмір отриманої БД (після імпорту), Мб
	№ експерименту	Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
Sones	1	78	23	50	1510 (RAM)
	2	78	23	51	1510 (RAM)
	3	79	22	50	1510 (RAM)
	4	77	24	50	1510 (RAM)
	5	77	23	50	1510 (RAM)
	6	77	23	50	1510 (RAM)
	7	78	22	50	1510 (RAM)
	8	78	22	50	1510 (RAM)
Neo4J	1	13	10	27	1308
	2	13	11	24	1308
	3	13	10	24	1308
	4	14	10	24	1308
	5	14	10	24	1308
	6	13	10	24	1308
	7	13	11	24	1308
	8	13	10	24	1308

Закінчення таблиці 4.2

Назва БД	Час (с)				Розмір отриманої БД (після імпорту), Мб
	№ експерименту	Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
DEX	1	20	13	25	1803
	2	21	13.5	26	1803
	3	20	13.1	25	1803
	4	20	14	26	1803
	5	20	13.2	25	1803
	6	20	13.2	25	1803
	7	21	13.5	25	1803
	8	20	13.1	25	1803
SQL Server	1	19	16.1	42	2835
	2	19	16.5	42.5	2835
	3	19	16	42	2835
	4	19	16.3	42	2835
	5	19	16.3	42	2835
	6	19	16.2	42	2835
	7	19	16.3	42.1	2835
	8	19	16.4	42	2835

Таблиця 4.3 – Результати при 4 000 000 вершин та 38 000 000 ребер

Назва БД	№ експерименту	Час (с)			Розмір отриманої БД (після імпорту), Мб
		Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
Sones	1	4150	301	622	5792 (RAM)
	2	4143	280	612	5792 (RAM)
	3	4165	290	612	5792 (RAM)
	4	4158	303	613	5792 (RAM)
	5	4164	301	612	5792 (RAM)
	6	4166	307	612	5792 (RAM)
	7	4159	306	611	5792 (RAM)
	8	4161	298	613	5792 (RAM)
Neo4J	1	2430	180	490	5197
	2	2430	183	489	5197
	3	2430	183	512	5197
	4	2430	179	495	5197
	5	2430	182	499	5197
	6	2431	179	497	5197
	7	2430	185	503	5197
	8	2430	181	497	5197

Закінчення таблиці 4.3

Назва БД	№ експерименту	Час (с)			Розмір отриманої БД (після імпорту), Мб
		Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
DEX	1	812	210	560	7238
	2	790	221	418	8512
	3	790	225	417	8512
	4	791	215	425	8512
	5	792	219	429	8512
	6	791	218	420	8512
	7	791	214	421	8512
	8	792	219	419	8512
SQL Server	1	2115	315	890	10586
	2	2113	305	860	10586
	3	2113	312	872	10586
	4	2113	307	892	10586
	5	2113	308	881	10586
	6	2113	309	887	10586
	7	2113	309	887	10586
	8	2113	307	889	10586

З отриманих результатів експерименту, який охоплював графові бази даних Sones, DEX та Neo4J на масштабах даних, складаючих 4 000 000 вершин та 38 000 000 ребер, виявлено цікаві особливості продуктивності.

Не дивлячись на те, що графова СУБД Sones виявилася менш швидкою порівняно з іншими графовими та реляційними системами, важливим є те, що вона все одно перевершує реляційну СУБД SQL Server за часом виконання запитів. Це свідчить про те, що, навіть в умовах меншої ефективності, графові

бази даних можуть виявитися вигідними у порівнянні з традиційними реляційними аналогами.

З іншого боку, графова СУБД Neo4J демонструє певні компроміси між часом імпорту бази даних та продуктивністю пошуку. Хоча вона може вимагати більше часу для імпорту, на великих обсягах даних, особливо при 4 000 000 вершин та 38 000 000 ребер, Neo4J відзначається надзвичайно швидким пошуком сусідів та перетином множини сусідів. Це зробило її однією з найефективніших графових СУБД у даному експерименті.

Також важливо відзначити, що СУБД Dex, хоч і показав часом швидший пошук перетину множини сусідів порівняно з Neo4J, загалом не дотягнув до швидкості роботи Neo4J у багатьох інших аспектах. Враховуючи отримані результати, можна зробити висновок, що графова СУБД Neo4J, не дивлячись на тривалий час імпорту, виявляється перспективним вибором для проектів з обробки великих обсягів даних та взаємозв’язаною структурою.

Таблиця 4.4 – Результати при 10 000 000 вершин та 90 000 000 ребер

Назва БД	№ експерименту	Час (с)			Розмір отриманої БД (після імпорту), Мб
		Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
Sones	1	–	–	–	–
	2	–	–	–	–
	3	–	–	–	–
	4	–	–	–	–
	5	–	–	–	–
	6	–	–	–	–
	7	–	–	–	–
	8	–	–	–	–

Закінчення таблиці 4.4

Назва БД	№ експерименту	Час (с)			Розмір отриманої БД (після імпорту), Мб
		Імпорт даних у БД	Пошук сусідів до заданого рівня	Пошук перетину множини сусідів	
Neo4J	1	3802	360	680	13215
	2	3805	382	685	13215
	3	3804	352	670	13215
	4	3802	360	681	13215
	5	3803	367	687	13215
	6	3802	362	681	13215
	7	3805	365	686	13215
	8	3801	363	684	13215
DEX	1	2045	241	607	18103
	2	2046	212	519	19400
	3	2018	208	518	19400
	4	2001	213	525	19400
	5	2019	215	521	19400
	6	2024	219	523	19400
	7	2015	224	524	19400
	8	2018	214	522	19400
SQL Server	1	7392	612	1398	26203
	2	7390	590	1312	26203
	3	7390	598	1410	26203
	4	7390	601	1352	26203
	5	7390	602	1352	26203
	6	7390	602	1357	26203
	7	7390	601	1354	26203
	8	7390	605	1356	26203

За результатами експерименту, де відсутність графової СУБД Sones була компенсована винятковими досягненнями DEX, виявлено, що остання виявилася першовиборною на даному обсязі даних. Навіть при великих обсягах, які викликали труднощі для Sones, DEX виявила себе фаворитом завдяки вражаючій продуктивності та швидкості виконання запитів.

DEX продемонструвала свою домінантність у кількох аспектах. Вона не тільки значно перемогла в імпорті даних, бувши швидше за Neo4J на 90% та реляційну СУБД SQL Server у 2 рази, але й виявилася більш ефективною в усіх типах пошуків. Її швидкість пошуку сусідів до заданого рівня перевищила інші графові та реляційні аналоги, з надзвичайним перевагою у 60% порівняно з реляційною СУБД SQL Server. Крім того, у важкому завданні пошуку перетину множини сусідів DEX виявилася ефективнішою, швидше за Neo4J на 20% та реляційну СУБД SQL Server у 2.2 рази.

Отже, можна визначити, що в умовах великих обсягів даних та наявності достатніх ресурсів, зокрема пам'яті на диску та RAM, графова СУБД DEX виходить на перший план, надаючи найкращі результати у всіх аспектах ефективності порівняно з конкурентами.

Висновки до розділу 4

Проведений експеримент над дослідженням часової ефективності графових баз даних у порівнянні з реляційною базою даних дозволяє зробити кілька важливих висновків. Здійснене порівняння різних графових баз даних, зокрема Sones, Neo4J та DEX, в контексті їхньої продуктивності та впливу кількості вершин на час виконання запитів, надає можливість краще зрозуміти їхню ефективність та обмеження.

Отримані результати свідчать про те, що графові бази даних в цілому є життєздатними та виявляються ефективними при опрацюванні великих обсягів даних, збільшуючи ефективність в рази, при збільшенні обсгів даних та їх взаємозв’язків. Однак, ретельний аналіз результатів дозволяє зробити висновок, що Sones GraphDB має обмежені можливості, особливо на великих обсягах

даних, оскільки зберігається лише в оперативній пам'яті та не досягає необхідної продуктивності порівняно з іншими базами даних.

Neo4J, незважаючи на свою популярність та різноманіття доступних API, також має свої обмеження, особливо коли мова йде про обробку великих підграфів. Ефективність Neo4J спостерігається лише у випадку невеликих обсягів даних, до 1,5 мільйонів об'єктів.

З іншого боку, DEX GraphDB виявила себе як більш перспективна графова база даних, особливо на великих обсягах даних, перевершуючи навіть реляційну СУБД SQL Server у плані часової ефективності. Її здатність працювати ефективно з обсягами даних більше 10 мільйонів вершин та 90 мільйонів ребер робить DEX GraphDB привабливим вибором для проектів з великою пов'язаністю даних.

Важливо враховувати, що різні графові бази даних взаємодіють по-різному з різними обсягами та характером даних. Деякі графові бази даних можуть використовувати кешування та інші механізми оптимізації, тоді як інші можуть виявити себе менш ефективними та непередбачуваними на великих обсягах даних. Такий індивідуальний підхід до вибору графової бази даних має вирішальне значення для успішної реалізації проєктів з обробки великої кількості пов’язаних даних.

Також ми бачимо як кількість вершин впливає на час пошуку сусідів до заданого рівня та перетину сусідів на прикладі часу роботи реляційної СУБД. Майже всі графові бази даних показують швидший час пошуку, порівняно з реляційними при меншому обсягу використаної пам’яті при розгортанні бази у СУБД. Тож можемо прийти до висновку, що при великій пов’язаності даних між собою – графові бази даних показують переваги у часі під час пошуку по даним та сусіднім до них зв’язкам.

Крім того, важливо розглядати, як саме графові СУБД оптимізують час виконання запитів. Вони можуть використовувати алгоритми оптимізації запитів для вдосконалення шляху виконання операцій та мінімізації часу виконання.

Паралельно з цими оптимізаціями, значущим є внутрішній механізм розпаралелювання, що дозволяє розділяти завдання на декілька низькорівневих

операцій, які виконуються паралельно. Велике значення має також здатність операційної системи та архітектури процесора впоратися із розпаралеленням, що допомагає оптимізувати час виконання запитів.

Треба додатково зазначити, що в подібних експериментах дбається про максимальну об'єктивність результатів через "холодний" запуск віртуальної машини перед кожним новим етапом експерименту. Це сприяє уникненню впливу кешування та оптимізації, що є критичним для досягнення максимально порівняних результатів у відповідних умовах.

ВИСНОВКИ

В даній роботі представленні дослідження часової ефективності графових СУБД порівняно з реляційними СУБД, можна прийти до декількох ключових висновків.

Виконано дослідження з аналізом результатів в порівнянні часових показників імпорту, запитів пошуку та розміру використаної пам’яті після розгортання бази даних. Були порівняно між собою графові СУБД: Neo4J, Sones, DEX та реляційну MS SQL Server.

Результати проведених експериментів, які включали аналіз продуктивності графових СУБД Sones, DEX та Neo4J, на обсягах даних від 16 000 вершин та 600 000 ребер до 4 000 000 вершин та 38 000 000 ребер, призвели до визначення ключових аспектів їхньої технічної ефективності.

На початковому етапі експерименту, де обсяг даних був менший, Sones, DEX та Neo4J не виявили переваг у швидкості виконання запитів та імпорту штампу бази даних порівняно з реляційною СУБД MS SQL Server. Зазначимо, що Neo4J вирізнялася подвійно швидшим імпортом, на 50% більш ефективним виконанням запитів пошуку сусідів та більш ніж вдвічі покращеною швидкістю пошуку перетину множини сусідів.

У подальших експериментах з великим обсягом даних (4 000 000 вершин та 38 000 000 ребер), Sones, хоч і менш швидка, виявила перевагу над реляційною СУБД SQL Server за часом виконання запитів, відображаючи, що графові бази даних можуть бути конкурентоспроможними у порівнянні з традиційними реляційними системами, навіть при меншій ефективності.

Neo4J, хоча вимагає більше часу для імпорту, відзначилася надзвичайно швидким пошуком сусідів та перетином множини сусідів, що робить її перспективним вибором для проектів з обробки великих обсягів даних та взаємозв’язаною структурою.

DEX, з своєю вражаючою продуктивністю, стала фаворитом при великих обсягах даних. Швидкість імпорту та всі типи пошуків (включаючи пошук

перетину множини сусідів) виявилися значно кращими, роблячи DEX найефективнішою графовою СУБД у даному експерименті.

Отже, при належній конфігурації ресурсів, графові СУБД, зокрема DEX та Neo4J, можуть ефективно конкурувати та вирішувати завдання обробки великих обсягів взаємозв’язаних даних порівняно з реляційними аналогами.

Загалом, вибір між графовими та реляційними базами даних повинен бути обґрунтованим і залежати від конкретних вимог та особливостей проекту. Графові бази даних можуть бути потужним інструментом для вирішення завдань, пов'язаних з великою кількістю взаємозв'язків у даних, але важливо враховувати їхні обмеження та оптимізації в конкретних сценаріях використання.

При розробці нового програмного забезпечення рекомендується розглядати використання реляційних СУБД, зокрема MS SQL Server, для проектів з невеликим обсягом даних та простою структурою. У випадку великих обсягів взаємозв’язаних даних та потреби в швидкому доступі до них, графові СУБД, такі як DEX та Neo4J, можуть бути оптимальними виборами. Для взаємодії з графовими базами даних рекомендується використовувати API, наприклад, DEX API або Neo4J API, які надають розширений функціонал для зручного доступу до графових даних.

Важливо провести додаткові дослідження, зокрема вивчити вплив різних типів запитів на часову ефективність кожної графової СУБД та оптимізацію запитів для кращої продуктивності. Також слід дослідити вплив конфігурації сервера на роботу графових СУБД. Ці дослідження допоможуть зрозуміти, яка СУБД найбільше відповідає вимогам конкретного проекту та як оптимально взаємодіяти з нею при розробці програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Джеймс Дж., Сміт П. Графові бази даних: Порівняльний аналіз та практичні застосування [Книга] / П. Сміт, Дж. Джеймс. – Видавництво "Техніка", 2020. – 120-145 с.

2) Кларк А., Браун Р. Порівняльний аналіз швидкості виконання запитів в графових та реляційних базах даних [Стаття] / Р. Браун, А. Кларк. – Журнал "Дослідження баз даних", 2019. – Сторінки: 45-60.

3) Офіційна документація Neo4J [Електронний ресурс]. Режим доступу: http://neo4j-docs.com.

4) Сміт Д., Харрісон М. Імплементація графових баз даних у великих корпоративних середовищах [Конференція] / М. Харрісон, Д. Сміт. – Конференція "Технології баз даних", 2018. – Сторінки: 210-225.

5) Реляційні та графові бази даних: технології порівняння [Онлайн-ресурс] DB-Engines. Режим доступу: http://db-engines.com/en/compare/relational-vs-graph.

6) Гусєв І., Львова Т. Графові бази даних у високонавантажених системах [Стаття] / І. Гусєв, Т. Львова. – Журнал "Інформаційні технології в бізнесі", 2017. – Сторінки: 78-95.

7) Житник О., Ковальчук В. Використання Neo4J для оптимізації пошуку перетину множини сусідів [Конференція] / В. Ковальчук, О. Житник. – Конференція "Пошук та аналіз даних", 2019. – Сторінки: 150-165.

8) Дані досліджень інтернет-аудиторії України [Електронний ресурс]. Режим доступу: http://www.inau.org.ua/analytics.

9) Stochastic diffusion search [Virtual Resource] // Wikipedia. Access Mode: URL: https://en.wikipedia.org/wiki/Stochastic_diffusion_search.

10) Компоненты сетевого приложения. Клиент-серверное взаимодействие і ролі серверів [Електронний ресурс]. Режим доступу: http://www.4stud.info/networking/lecture5.html.

11) Шинкаренко В.І., Куроп'ятник О.С, Забула Г.В, Пєтін Д.О., Лукін Є.В. Якість програмного забезпечення та тестування [Текст]: методичні вказівки до

лабораторних / уклад.: В.І. Шинкаренко, О.С. Куроп'ятник, Г.В. Забула, Д.О. Пєтін, Є.В. Лукін, Дніпропетр, нац. ун-т. залізн. трасоп. ім. акад. В. Лазаряна. – Вид-во ПФ «Стандарт-Сервіс», 2018. – 50 с. – 43с.

12) DEX Officer Inc — зручний і багатофункціональний сервіс для аналізу інтернет-сайтів та додатків [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Google_Analytics.

13) Проектирование по ис в Rational Rose Диаграмма варіантів використання [Електронний ресурс]. Режим доступу: https://studfiles.net/preview/5187971/page:3/.

14) Erlonger C.M. [Текст] Work with MySql, MS SQL Server, Examples / C.M. Erlonger — USA, 248. -301 с.

15) ГОСТ 19.701-90 ЕСПД. Схеми, алгоритми та структури даних.

16) Тестування програмного забезпечення [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення.

17) Гембл Т., Браун К. Особливості графових баз даних у сфері великих даних [Стаття] / К. Браун, Т. Гембл. – Журнал "Біг Дата і Аналітика", 2020. – Сторінки: 110-125.

18) Офіційна документація Sones. Режим доступу: http://sones-docs.com. [Електронний ресурс].

19) Петерсон Е., Веллер М. Використання графових баз даних у великих соціальних мережах [Стаття] / М. Веллер, Е. Петерсон. – Журнал "Соціальна інформатика", 2019. – Сторінки: 200-215.

20) Под капотом у Stopwatch [Електроний ресурс] // Хабр. Тип доступа: https://habr.com/ru/post/226279/.

21) NLog tutorial [Електроний ресурс] // GitHub. Access Mode: URL: https://github.com/nlog/nlog/wiki/Tutorial.

22) Brown function as benchmark [Електроний ресурс] // BenchmarkFcns. Access Mode: URL: http://benchmarkfcns.xyz/benchmarkfcns/brownfcn.htm.

ДОДАТОК А

Тези XVII МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

Дослідження часової ефективності графових баз даних

Баша П. О., Шинкаренко В. І., Український державний університет науки і технологій, Україна

На сьогодні проблема використання неоптимальних баз даних стає все більш актуальною в контексті зростаючого обсягу та складності пов'язаних даних. Реляційні бази даних, які довгий час були стандартом для зберігання та обробки інформації, стикаються з обмеженнями у випадках, коли важлива структура зв'язків між об'єктами. Зокрема, у великих проєктах зі значною кількістю пов'язаних даних реляційні бази даних можуть стати ефективними з точки зору продуктивності та часових характеристик. У зв'язку з цим, графові бази даних визнаються як потенційна альтернатива, спроможна ефективно вирішувати завдання, пов'язані з глибоким аналізом зв'язків та великим обсягом пов'язаних даних.

Представлена робота присвячена дослідженню часової ефективності графових баз даних, як важливого аспекту управління та обробки пов'язаних між собою даних. В сучасних інформаційних технологіях управління та обробка даних є важливою задачею, і вибір оптимальної системи управління базами даних є критичним завданням для забезпечення критичних характеристик інформаційних систем.

На даний момент реляційні та NO-SQL бази даних широко використовуються і розглядаються як оптимальний варіант для обробки даних у великих програмних проєктах. Однак, графові бази даних набирають популярність через свою спрямованість на представлення та оптимізацію структури зв'язків між об'єктами.

Дослідження включає в себе вибір чотирьох різних графових систем керування баз даних ¬– Sones GraphDB (графова СКБД), Neo4J GraphDB (графова СКБД), DEX GraphDB (графова СКБД) та Microsoft SQL Server 2012 (реляційна СКБД) – для подальшого порівняння.

Для вимірювання часової ефективності використовується віртуальна машина та формується база даних з різною кількістю вершин та зв'язків між ними. До кожної бази даних виконуються декілька запитів, визначається середній час обробки запитів. Цей підхід дозволяє нам докладно дослідити вплив розміру та складності графів на часові показники обробки даних у кожній системі.

Метою даної роботи є висвітлення можливості графових баз даних та їх ефективність в умовах проєктів з великою кількістю пов'язаних даних на основі ґрунтовних експериментальних досліджень.

Згідно з отриманими результатами дослідження, виокремлюється тенденція до геометричного збільшення переваг деяких графових баз даних при зростанні кількості вершин. При аналізі роботи баз даних з різною кількістю вершин доводиться, що на певному етапі реляційні бази даних можуть виявити вищу швидкість, але з подальшим збільшенням обсягу даних перевага графових баз даних стає вкрай виразною.

Зокрема, при роботі з 16 000 вершинами реляційна база даних може виявити перевагу над деякими графовими базами. Проте, при збільшенні обсягу даних, наприклад, до 100 000 вершин, ми спостерігаємо, що база даних Sones знову поступається в часовій ефективності, тоді як Neo4J та DEX продовжують демонструвати конкуренто–спроможність.

Найцікавіше, що при досягненні 4 000 000 вершин та 38 000 000 ребер Neo4J не може надати адекватну відповідь у прийнятний часовий проміжок, тоді як DEX виявляється найшвидшим рішенням, перевершуючи реляційну базу даних SQL Server в 8.3 рази.

Результати експериментів засвідчують значний потенціал графових систем керування баз даних, що відповідає тенденціям у напряму Big Data.

ДОДАТОК Б

Технічне завдання

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і
технології
Анатолій РАДКЕВИЧ

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01360-01-ЛЗ

Завідувач кафедрою КІТ
_____ Вадим ГОРЯЧКІН

Керівник розробки, професор
_____ Віктор
ШИНКАРЕНКО

Виконавець
_____ Павло БАША

Нормоконтролер, асистент
_____ Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО
44165850.01360-01-ЛЗ

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Технічне завдання

Листів 24

2024

АНОТАЦІЯ

Документ 44165850.01360-01 «Програмна система Database Timelaps Test. Технічне завдання». входить до складу програмної документації для програмного інструменту дослідження та аналізу часової ефективності графових баз даних.

Розроблена система дозволяє визначити доцільність застосування графових баз даних на певних розмірах вибірки пов’язаних між собою.

В даному документів представлено специфікацію програмного інструменту, а також область застосування програмного продукту, його призначення, стадії та строки виконання проекту та порядок контролю і приймання.

ЗМІСТ

ВСТУП

Найменування проекту: «Програмна система Database Timelaps Test».

Основна термінологія.

База даних (БД) – це організована колекція даних, яка зберігається та обробляється комп'ютерною системою таким чином, щоб можна було ефективно виконувати операції додавання, зміни та вилучення інформації. База даних є основою для зберігання та організації структурованих даних, які можуть включати текст, числа, зображення, звуки та інші види інформації.

СУБД (Система управління базами даних) – це програмне забезпечення, яке дозволяє створювати, зберігати, оновлювати та взаємодіяти з базами даних. СУБД надає зручний інтерфейс для визначення структури бази даних, виконання операцій додавання, зміни та вибірки даних, а також забезпечує захист інформації та оптимізацію доступу до даних.

Графова база даних – це тип бази даних, яка використовує графову модель даних для представлення і зберігання інформації. У графовій базі даних дані представлені у вигляді вузлів (вершин) та зав’язків (ребер) між цими вузлами. Кожен вузол може мати атрибути, які визначають його властивості, а зв'язки вказують на взаємозв'язки між вузлами. Графові бази даних ефективно використовуються для вирішення завдань, пов'язаних з аналізом мережевих взаємозв'язків, таких як соціальні мережі, транспортні системи та рекомендаційні системи.

Реляційна база даних - це тип бази даних, яка використовує реляційну модель даних для організації і управління інформацією. Дані в реляційних базах даних представлені у вигляді таблиць, де кожен рядок представляє кортеж (запис), а кожний стовпець - атрибут (поле). Зв'язки між таблицями встановлюються за допомогою ключів. Реляційні бази даних широко використовуються у бізнесі та інших сферах для забезпечення структурованого та ефективного зберігання даних, а також для виконання операцій обробки даних за допомогою мови SQL.

Список суміжності - це структура даних в графовій теорії, що представляє граф, перераховуючи для кожної вершини всі вершини, з якими вона безпосередньо пов'язана, разом із зв'язками між ними. Цей метод ефективно вказує сусідство вершин і дозволяє легко визначати структуру графа та швидко знаходити інформацію про зв'язки між його елементами.

Database Timelaps Test – назва системи, яка в перекладі означає тестування часових метрик баз даних.

Потреба у розробці даного програмного інструментарію виникла внаслідок необхідності проведення досліджень та відсутності аналогів для таких досліджень. Це програмне забезпечення призначене для використання в дослідницьких центрах, наукових установах тощо з метою узагальнення часової ефективності графових баз даних у порівнянні з реляційними на різних об’ємах пов’язаних даних.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Основою для розробки є наказ ректора Українського державного університету науки і технології Радкевич А.В. «Про затвердження тем та призначення керівників дипломних проектів» №1113 ст від 02.11.2022 року.

Тема проекту: «Дослідження часової ефективності графових баз даних».

Керівник дипломного проекту: професор Шинкаренко В.І.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

Програмна система призначена для дослідження часової ефективності графових баз даних у порівнянні з реляційними на різних об’ємах пов’язаних даних;

2.2 Експлуатаційне призначення

Програмна система дозволить в подальшому створювати більш ефективні системи та програмне забезпечення, ураховуючи цільові об’єми інформації, з якими буде працювати ПЗ.

3 ВИМОГИ ДО ПРОГРАМИ АБО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Досліди пов’язані з покращенням роботи існуючих та створенням нових біонічних алгоритмів актуальні і сьогодні. Основним напрямком даних досліджень – створення різноманітних гібридів та модифікацій задля пришвидшення роботи виконання поставлених задач та подолання певних проблем.

3.1.1 Вимоги до можливостей програмного інструментарію

Нижче наведено перелік основних вимог до функціоналу який повинен бути присутнім в програмній системі:

– вибір дата-сету (заповненної бази даних), над якими буде екперимент;

– введення необхідних параметрів для авторизації в системі СУБД;

– введеня типу запиту, часова метрика якого буде вимірюватись;

– вибір граничного розміру даних, який повинен бути використаний в конкретному експерименті;

– отримання файлу з записами (логуванням) інформації та результатів екперименту;

– можливість обрати тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД.

3.1.2 Вимоги до вхідних даних

Всі вхідні параметри та вимоги до них наведено нижче:

– вибір дата-сету (заповненної бази даних), над якими буде екперимент;

– введення необхідних параметрів для авторизації в системі СУБД;

– введеня типу запиту, часова метрика якого буде вимірюватись;

– вибір граничного розміру даних, який повинен бути використаний в конкретному експерименті;

– можливість обрати тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД

3.1.3 Вимоги до вихідних даних

Усі вихідні дані та вимоги до них наведено нижче:

– текстові повідомлення системи (табл. 3.1);

– часові показники роботи певного запиту до БД – ціле, невід'ємне число, в мілісекундах та тіках процесору;

– показники розміру імпортованої бази даних;

– показники часу імпорту певного дата-сету – ціле, невід'ємне число, в мілісекундах та тіках процесору.

3.2 Вимоги до надійності

Програмний продукт повинен відповідати наступним вимогам щодо надійності:

– резервна копія тексту програми має бути збережена на окремому носії даних і в репозиторії;

– у випадку введення користувачем некоректних даних система повинна інформувати користувача та пропонувати ввести правильні дані;

– при збоях система повинна повідомляти користувача про наявність та тип помилок;

– система повинна вести логування роботи програмного додатку.

3.3 Умови експлуатації

Під час експлуатації програмного додатку необхідне дотримання наступних вимог:

– користувач має володіти базовими навичками роботи з комп'ютером, а також має бути ознайомленим із керівництвом користувача;

– експлуатація розробленого продукту має відбуватись в приміщеннях, які відповідають нормам, щоб люді було ну так, нормально, щоб непогано було;

– усі складові ЕОМ мають відповідати нормам, визначеними нормативними документами.

3.4 Вимоги до складу і параметрів технічних засобів

Для повноцінного функціонування розроблюваної системи мають виконуватись наступні вимоги:

– наявність монітора або іншого електронного дисплею;

– наявність маніпулятора типу «миша»;

– не менше 5 Мб вільного місця на запам’ятовувальному пристрої;

– процесор з тактовою частотою не менше 1 ГГц;

– наявність джостику для управління літальним апаратом;

– 512 Мб оперативної пам’яті;

– користувач має володіти базовими навичками роботи з комп’ютером, а також має бути ознайомленим із керівництвом користувача;

– наявність зчитувальних пристроїв зовнішніх носіїв або підключення до мережі Інтернет – для встановлення програмного додатку.

3.5 Вимоги до інформаційної і програмної сумісності

Програмна система має виконуватись на ЕОМ з дотриманням наступних програмних вимог:

– операційна система Windows – програмний додаток створений на платформі Java Enterprise;

– має бути інстальований JDK (середовище виконання);

– має бути інстальоване програмне забезпечення для можливості відкриття текстових файлів формату .txt – для перегляду файлів логування роботи програми.

3.6 Вимоги до маркування і упаковки

У випадку, коли транспортування програмного продукту виконується через Інтернет-мережу, застосунок має бути розміщено на захищеному ресурсі, разом із необхідною документацією.

В тому випадку, коли транспортування розробленої системи виконується на окремому зовнішньому носії даних, необхідно щоб продукт був упакованим в упаковку захищену від механічних та кліматичних пошкоджень.

На упаковці повинна викладатись наступна інформація:

– назва продукту;
– номер версії;
– розробник;
– місце розробки
– апаратні та програмні вимоги;
– рік упаковки.

Приклад маркування зображено на рисунку 3.1.

3.7 Вимоги до транспортування і зберігання

В тому випадку, якщо транспортування програмного додатку виконується через Інтернет ресурс, даний ресурс має бути забезпеченим захисним функціоналом та обмеженим доступом для інших користувачів.

Доступ до ресурсу має бути отриманим в результаті проходження авторизації користувачем.

В тому випадку, коли вид транспортування – фізичний носій даних, необхідне дотримання наступних вимог:

– зберігання має бути в сухому захищеному від потрапляння прямих променів сонця місці;

– носій має бути захищеним від фізичних, механічних та хімічних ушкоджень;

Слід також відмітити обов’язкову наявність резервної копії фізичного носія.

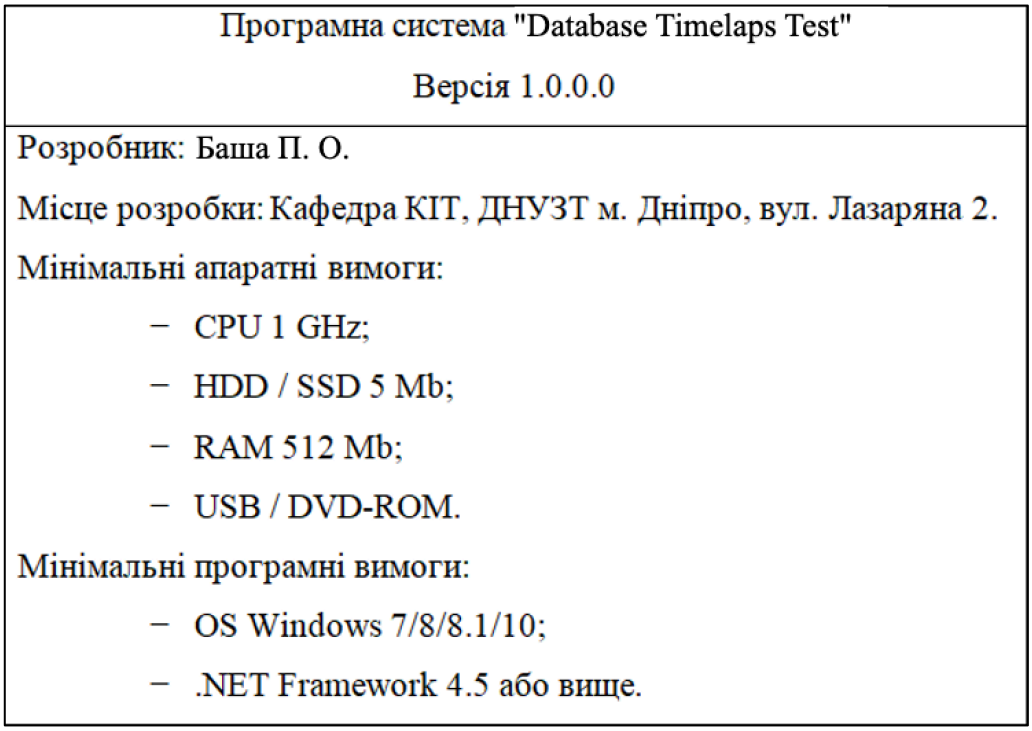


Рисунок 3.1 – Приклад маркування дистрибутивного носія даних розробленої програмної системи

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмна документація повинна складатись з таких документів:

- технічне завдання;
- робочий проект;
- специфікація;
- текст програми;
- опис програми;
- керівництво користувача. Керівництво дослідника.

Наведена вище програмна документація повинна задовольняти вимогам державних стандартів оформлення програмних документів.

5 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії та етапи розробки програмного продукту наведені представлені у вигляді таблиці (табл. 6.1)

Таблиця 6.1 – Стадії та етапи розробки

Стадія розробки	Етап розробки	Термін виконання
Технічне завдання (ТЗ)	Постановка задачі	10.2023
	Огляд літературних джерел та аналіз наявності програмного інструментарію	10.2023
	Розробка вимог до програми	10.2023
	Затвердження технічного завдання	11.2023
Робочий проект	Розробка і програмування логіки інструментального додатку	11.2023
	Відлагодження розробленого додатку	12.2023
Впровадження	Розробка програмної документації	01.2024

6 ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

Контроль за виконання роботи здійснює керівник дипломної роботи – професор кафедри КІТ Шинкаренко В.І.

Прийом виконаної роботи здійснюється уповноваженої комісією.

ДОДАТОК В

Технічне завдання

ЗАТВЕРДЖУЮ
Проректор
Українського державного
університету науки і технології
Анатолій РАДКЕВИЧ

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Робочий проєкт

ЛИСТ ЗАТВЕРДЖЕННЯ

44165850.01360-01-ЛЗ

Завідувач кафедрою КІТ
_____ Вадим ГОРЯЧКІН

Керівник розробки, професор
_____ Віктор
ШИНКАРЕНКО

Виконавець
_____ Павло БАША

Нормоконтролер, асистент
_____ Світлана ВОЛКОВА

2024

ЗАТВЕРДЖЕНО
44165850.01360-01

ДОДАТОК Г

Технічне Завдання

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Специфікація

Листів 2

2024

СПЕЦИФІКАЦІЯ

Позначення	Найменування	Примітка
Документація		
44165850.01360-01-ЛЗ	Лист затвердження	
44165850.01360-01	Специфікація	
44165850.01360-01 12 01-ЛЗ	Лист затвердження	
44165850.01360-01 12 01	Текст програми	
44165850.01360-01 13 01-ЛЗ	Лист затвердження	
44165850.01360-01 13 01	Опис програми	
44165850.01360-01 І3 01-ЛЗ	Лист затвердження	
44165850.01360-01 І3 01	Керівництво користувача. Керівницітов дослідника	

ЗАТВЕРДЖЕНО
44165850.01360-01 13 01

ДОДАТОК Ґ

Технічне Завдання

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Опис програми

Листів 25

2024

АНОТАЦІЯ

Документ 44165850.01360-01 13 01 «Програмна система Database Timelaps Test». Опис програми» входить до складу програмної документації для програмного інструменту дослідження та аналізу часової ефективності графових баз даних.

Розроблена система дозволяє дослідити часову ефективность графових баз даних у порівнянні з реляційними на різних об’ємах пов’язаних даних та визначити доцільність використання для конкретного ПЗ.

В даному документі наведено опис програми та її функціональні можливості. Система розроблена у середовищі інтегрованої розробки IntelliJ IDEA Community із застосуванням мови програмування Java.

ЗМІСТ

1 ЗАГАЛЬНІ ВІДОМОСТІ

Повна назва проекту: програмна система «Database Timelaps Test».

Для функціонування програмного додатку необхідна наявність платформи Java SE версії не нижче 21.0.1.

Мова розробки – Java.

При розробці використовувався програмний інструментарій – IntelliJ IDEA Community 2020.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Програмна система «Database Timelaps Test» надає наступні можливості:

Програмна система має надавати наступні можливості:

– вибір дата-сету (заповненної бази даних), над якими буде екперимент;

– введення необхідних параметрів для авторизації в системі СУБД;

– введеня типу запиту, часова метрика якого буде вимірюватись;

– вибір граничного розміру даних, який повинен бути використаний в конкретному експерименті;

– отримання файлу з записами (логуванням) інформації та результатів екпериментy;

– можливість обрати тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД.

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

3.1 Алгоритми програми

Основний алгоритм загальної роботи системи наведено нижче:

– дослідник (далі користувач) обирає потрібну базу даних;

– користувач запускає віртуальну машину, на якій обирає потрібну базу даних та запускає її;

– відкривається візуальне вікно-браузер для вводу необхідних даних користувача для доступу для дата-сету;

– визначається тип бази даних, та запускається зв’язок з нею через розроблене ПЗ для досліджень;

– вводиться тип запуту для тестування та всі необхідні параметри для тестування запиту;

– створюється штамп результатів та записується у певний файл.

Алгоритм роботи зв’язку розробленого ПЗ зображено у вигляді Блок-схеми (рис 3.1) :

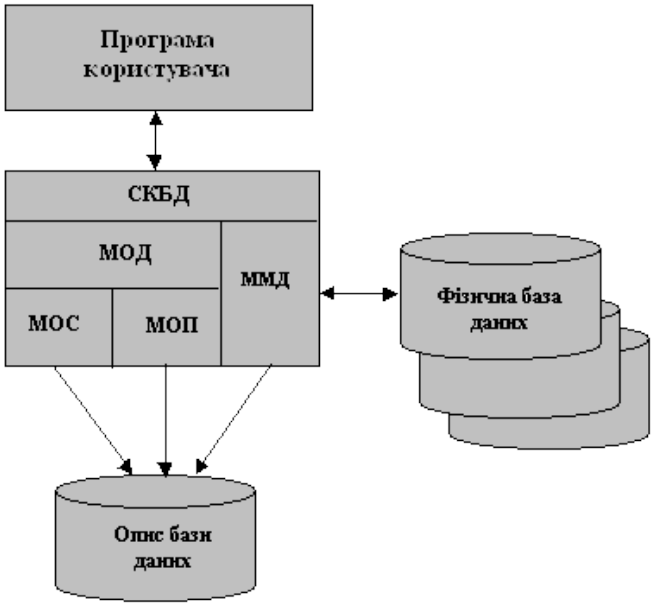


Рисунок 3.1 – Блок-схема взаємодії фізичної БД з СКБД

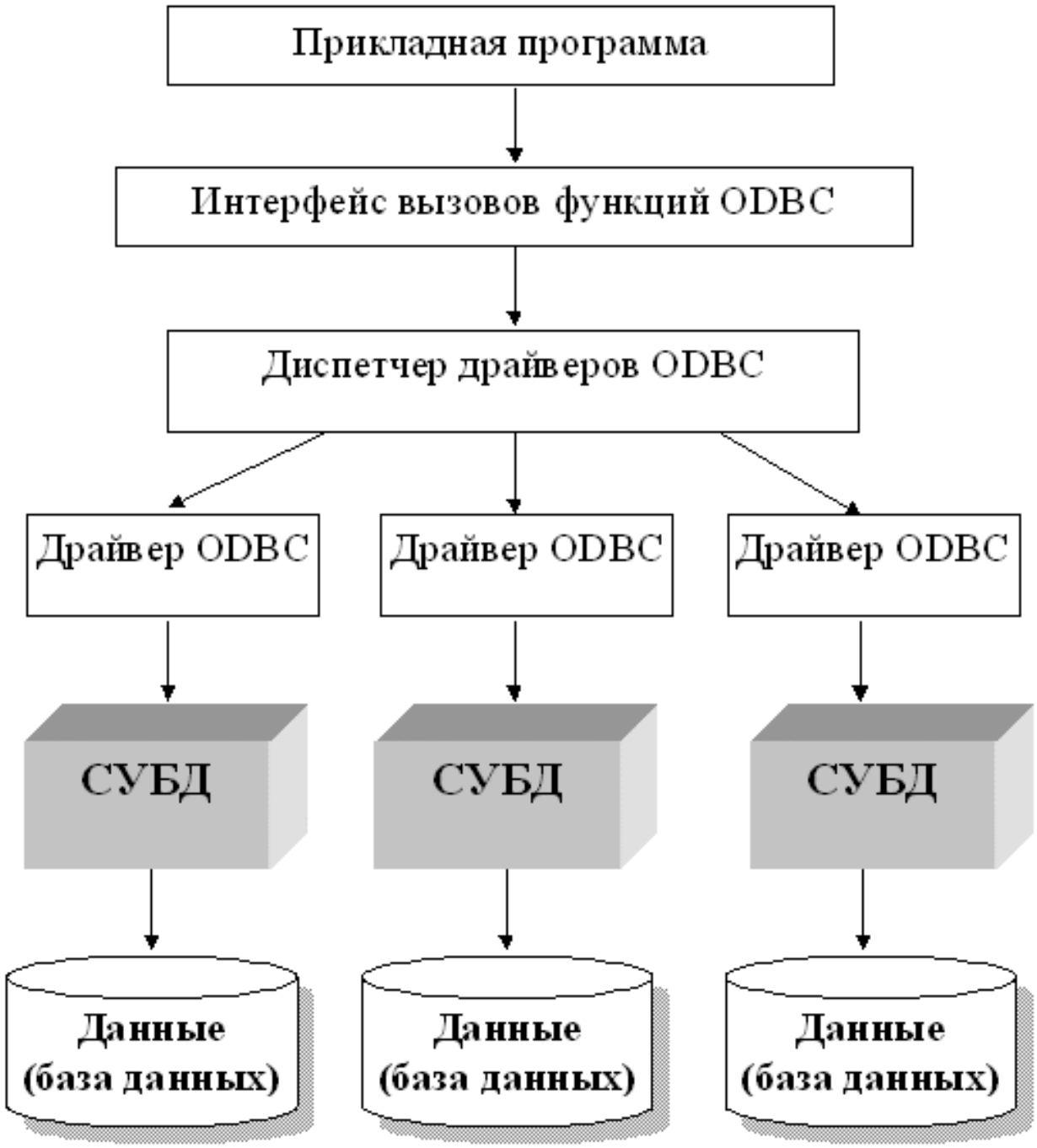


Рисунок 3.2 – Схема відображення блоків різних СУБД в віртуальній машині

У ході розгортання віртуальної машини для запуску графової бази даних (ГБД) було виконано ряд технічних заходів з метою оптимізації та забезпечення стабільності системи за схемою інтегрування СУБД на віртуальній машині (рис. 3.2):

– налаштування Віртуальної Машини: визначено та налаштовано ресурси, зокрема процесор, оперативну пам'ять та диск, для забезпечення високої продуктивності;

– включено віртуалізацію в налаштуваннях віртуальної машини для оптимального використання ресурсів.

– операційна Система: обрано оптимізовану операційну систему, сумісну з ГБД та підтримуючу необхідні залежності;

– налаштування Java Virtual Machine (JVM): задано параметри запуску JVM, включаючи використання пам'яті (-Xmx) та інші опції, щоб забезпечити ефективну роботу ГБД;

– інсталяція та Конфігурація ГБД: виконано процес встановлення графової бази даних згідно з інструкціями поставщика;

– налаштовано параметри з'єднань для ГБД, визначивши адресу сервера та інші необхідні деталі;

– мережева Конфігурація: створено та налаштовано віртуальну мережу з метою забезпечення доступу до ГБД з інших систем;

– моніторинг та оптимізація: застосовано інструменти моніторингу для постійного контролю за ресурсами віртуальної машини та продуктивністю ГБД;

– виконано оптимізацію параметрів ГБД та віртуальної машини з урахуванням результатів моніторингу.

Цей підхід дозволяє ефективно використовувати ГБД на віртуальній машині, забезпечуючи необхідну стабільність та продуктивність системи.

3.2 Структура програми

Програмний продукт має просту архітектуру, проте за рахунок єдиної відповідальності, застосованими шаблонам програмування, такими як абстрактна фабрика та стратегія, має змогу швидко розширюватись та змінювати певні елементи іншими із мінімальними змінами коду.

Нижче наведено таблицю основних класів та інтерфейсів для розробленого додатку (табл. 3.1).

Таблиця 3.1 – Основні класи та інтерфейси

Назва елементу	Опис
DatabaseTester	Відповідає за тестування часової ефективності графових та реляційних баз даних.
IDatabaseApi	Інтерфейс який дає контракт використання будь якої бази даних. Послідовники GraphDatabase та RelationalDatabase визначають типи баз даних, з якими система взаємодіє. GraphDatabase відповідає за графові бази даних, тоді як RelationalDatabase представляє реляційні бази. Ці класи надають необхідний інтерфейс для виконання запитів та управління даними відповідного типу.
QueryExecutor	Відіграє роль виконавця запитів до баз даних. Він отримує SQL- або графові запити від інших частин системи та відповідає за їхнє виконання, повертаючи результати у вигляді ResultSet.
ResultSet	Служить для представлення результатів виконаних запитів до бази даних. Його завдання - забезпечити структуроване представлення даних, отриманих в результаті виконання запитів.
ILogger	Визначає загальний інтерфейс для логування подій та даних в системі. Цей інтерфейс дозволяє різним класам системи взаємодіяти із системою логування.
FileLogger та ConsoleLogger	Реалізують інтерфейс ILogger і відповідають за логування інформації у файлі та консолі відповідно. Ці класи забезпечують різні методи виведення інформації.

Таблиця 3.1(продовження) – Основні класи та інтерфейси

ExperimentResult	Клас, що представляє результати конкретного експерименту. Він містить інформацію про часові показники, статистику та інші деталі, які можуть бути важливими для подальшого аналізу.
DataWarehouse	Клас визначає сховище даних, де зберігаються результати експериментів. Цей клас дозволяє управляти та зберігати експериментальні дані для подальшого використання та аналізу.
DataStamp	Клас відповідає за створення або імпорт штампу даних, який використовується для налаштування експерименту.
Node, Edge, Table, Column, DataType	Визначають основні елементи, що використовуються для представлення структури даних у графових та реляційних базах даних.
GraphData та RelationalData	Визначають структуру та особливості графових та реляційних даних відповідно. Вони представляють об'єкти, з якими буде взаємодіяти система.
DataBaseTransformator	Класо, який відповідає за трансформацію даних. Він може забезпечити конвертацію даних з одного формату до іншого для оптимізації експерименту.

3.3 Взаємодії класів системи

Під час створення архітектури системи було отримано наступну діаграму класів та інтерфейсів (рис. 3.2). Слід урахувати що на діаграмі зображено абстрактний шар загальної системи необхідної для її повноцінного функціонування.

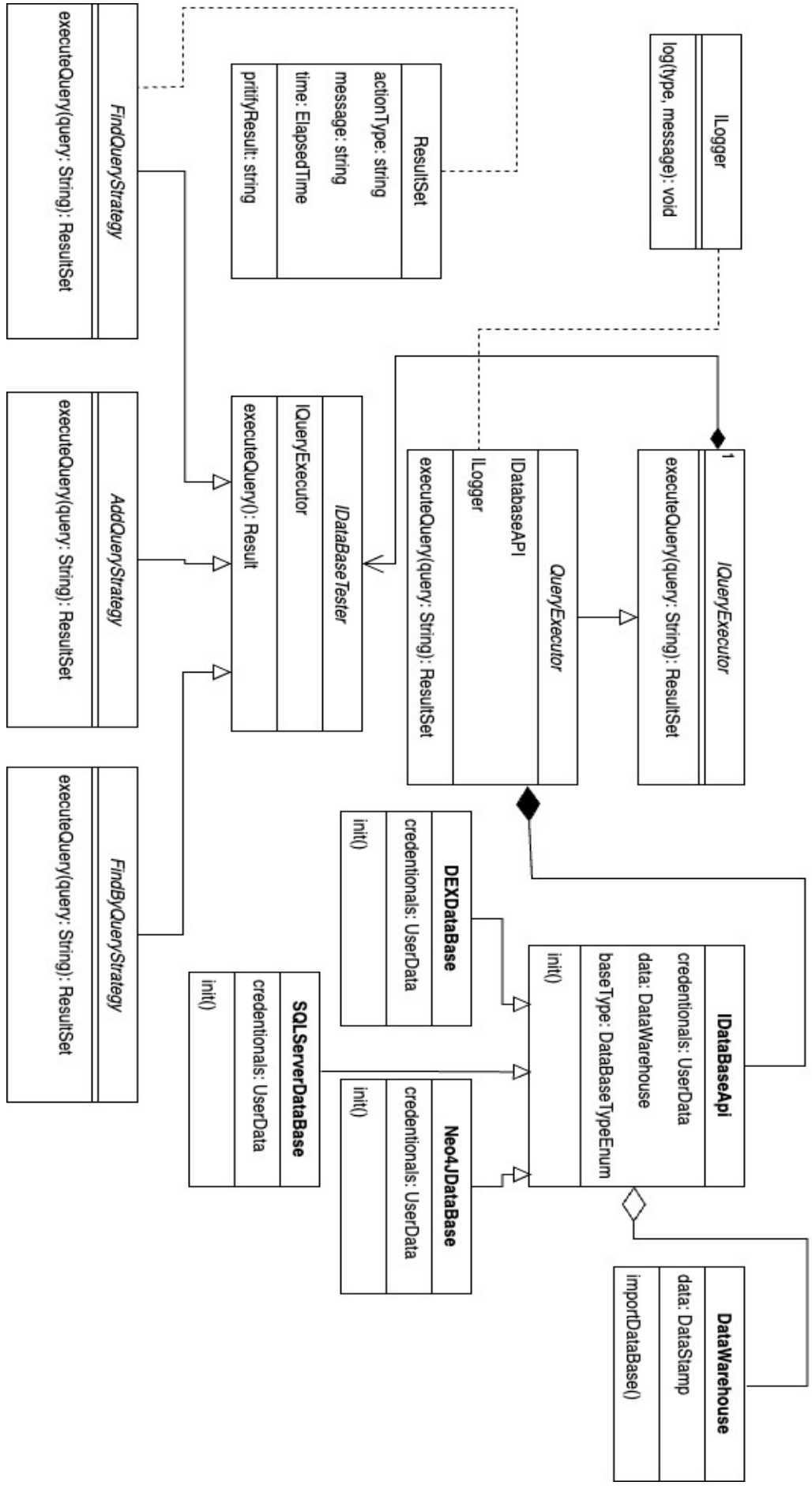


Рисунок 3.2 – UML схема взаємодії основних елементів системи

За рахунок використання поліморфізму було отримано наступні ієрархічні зв’язки:

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні реалізовувати інтерфейс IDatabaseApi;

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні успадковувати клас BaseEntryDatabase;

– усі варіанти логування повинні реалізовувати інтерфейс ILogger;

– інтерфейс IDataBaseTaster повинен використовувати тільки реалізації інтерфейсу IDatabaseApi для відтворення запитів до бази даних (рис 3.5).

Ці відносини забезпечують стабільність та можливість розширення системи, дозволяючи легко додавати нові компоненти чи змінювати існуючі, при цьому забезпечуючи відповідність до визначених інтерфейсів.

На малюнках 3.2 – 3.6 показано стретегії взаємодії класів.

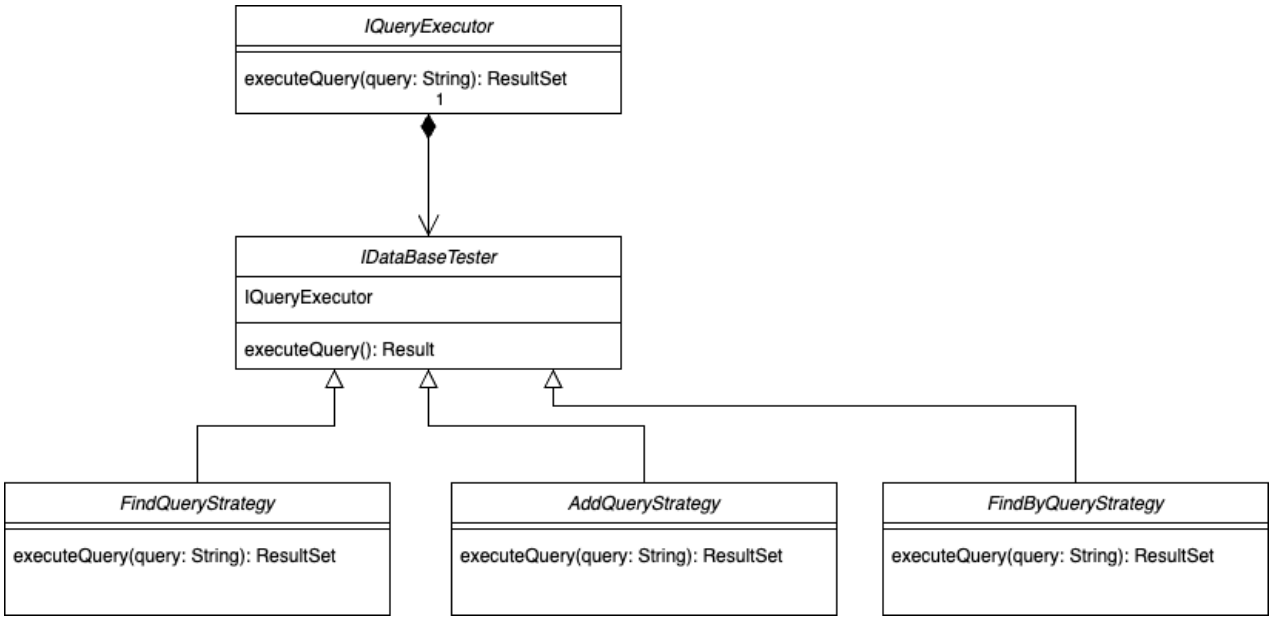


Рисунок 3.3 – Приклад використання спрощеної версії шаблону «абстрактна фабрика»

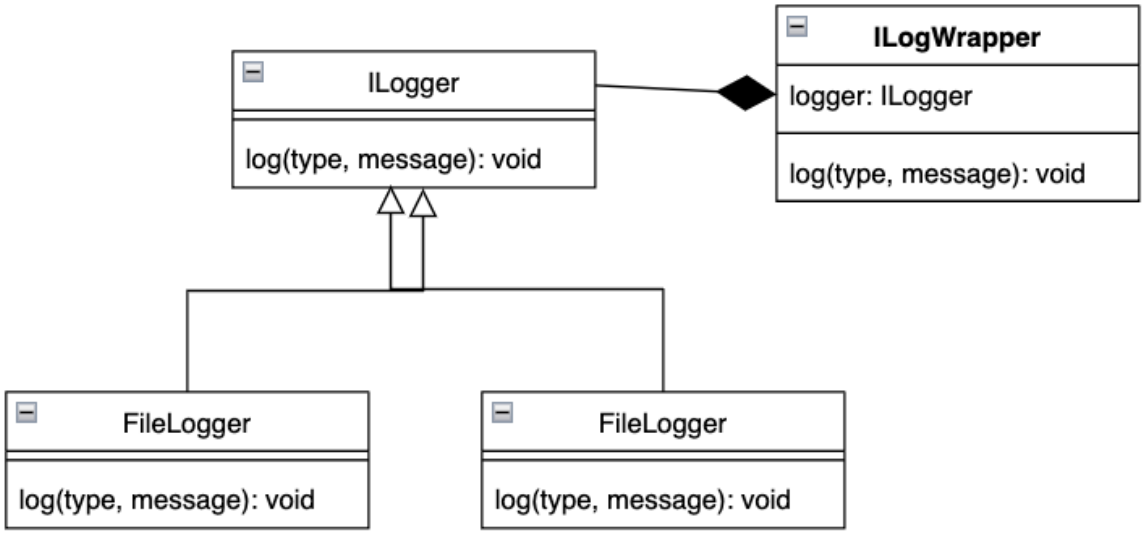


Рисунок 3.4 – приклад застосування шаблону Decorator

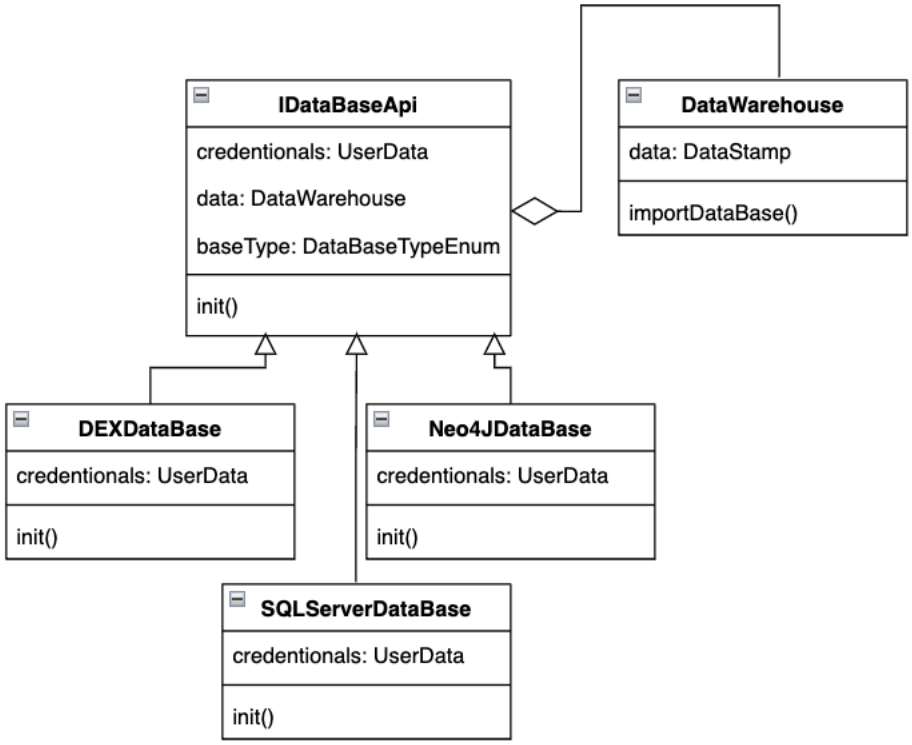


Рисунок 3.5 – приклад застосування шаблону «Абстрактний метод»

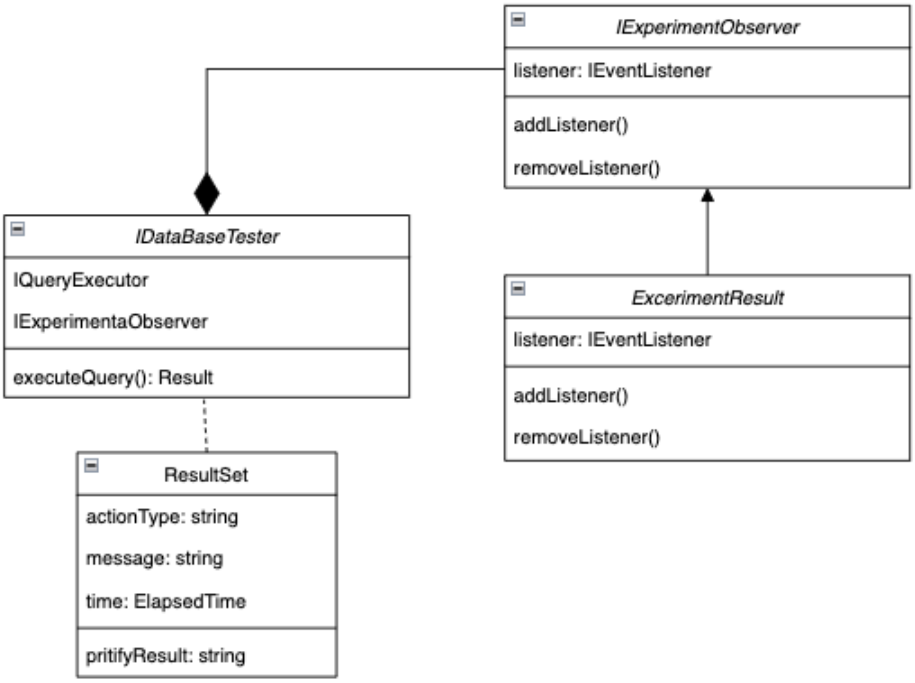


Рисунок 3.6 – приклад застосування шаблону «Observer»

4 ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Під час розробки програмної системи «Database Timelaps Test» використовувалось наступне технічне забезпечення:

- процесор Intel Corei7-4710HQ;
- 16 Гб оперативної пам’яті типу DDR4;
- накопичувач типу SSD 460 GB;
- клавіатура;
- маніпулятор типу «мишечка»;
- дисплей з розширенням 1920 x 1080.

Апаратне забезпечення:

- операційна система Windows 10 версії 1903;
- Java SE 21.0.1;
- Docker Desktop (virtual machine);
- MariaDB, MS SQL Server 2016;
- DEX Database API;
- Sones Database API;
- Neo4j Database API + Neo4j Browser.

5 ВИКЛИК І ЗАВАНТАЖЕННЯ

Для того аби розпочати роботу з системою необхідно виконати наступні пункти:

– переконатись, що на системі встановлено Java JDK (бажано останньої версії), Docker Desktop, Neo4J driver DB, Neo4J Browser, Sones driver DB, DEX driver DB;

– якщо файли програми «Database Timelaps Test» вже наявні на ЕОМ перейти до наступного пункту. В тому випадку, якщо програмного застосунку не має на ЕОМ необхідно скачати їх з хмарного сховища або ж з відповідного зовнішнього носія даних;

– відкрити папку з розташуванням файлів програми;

– відкрити файл «graph_timelaps_test.exe»;

– розпочати використання програми, слідуючи підказкам та запитам програми;

– для перегляду файлу з інформацією отриманою під час логування необхідно відкрити текстовий файл «results.txt».

Для завершення роботи з програмою необхідно закрити консольне вікно, або ж ввести комбінацію клавіш Ctrl + Z.

6 ВХІДНІ ДАНІ

Нижче наведено описову інформацію вхідних даних:

– обраний дата-сет (заповненна база даних), над яким буде екперимент;

– параметри доступу до бази даних (авторизації) в системі СУБД;

– тип запиту, часова метрика якого буде вимірюватись;

– граничний розмір даних, який повинен бути використаний в конкретному експерименті;

– тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД.

Всі дані, які вводяться або обираються, повинні перевірятися; у разі помилки повинна бути показана відповідна підказка.

7 ВИХІДНІ ДАНІ

Вихідною інформацією програмного інструменту є проміжні повідомлення, що стосуються проведення конкретного експерименту, та файл, що містить журнали роботи додатку. Зміст файлу еквівалентний повідомленням, які надходять користувачеві при використанні користувацького інтерфейсу. Ці повідомлення містять наступну інформацію:

- загальний час витрачений на роботу імпорту бази даних;
- час витрачений на запит пошуку сусідніх вершин до заданого рівня;
- час витрачений на запит пошуку перетину множини сусідів;
- розмір імпортованої бази даних.

8 ОПИС ІНТЕРФЕЙСУ КОРИСТУВАЧА

Для запуску програмного продукту, спочатку необхідно виконати пункти розділу 5.

Спочатку необхідно запустити віртуальну машину з необхідним екземпляром бази даних на ній. В нашому випадку, в ролі цієї віртуальної машини є Docker Container з відповідною базою даних, яку показано на рисунку 8.1.

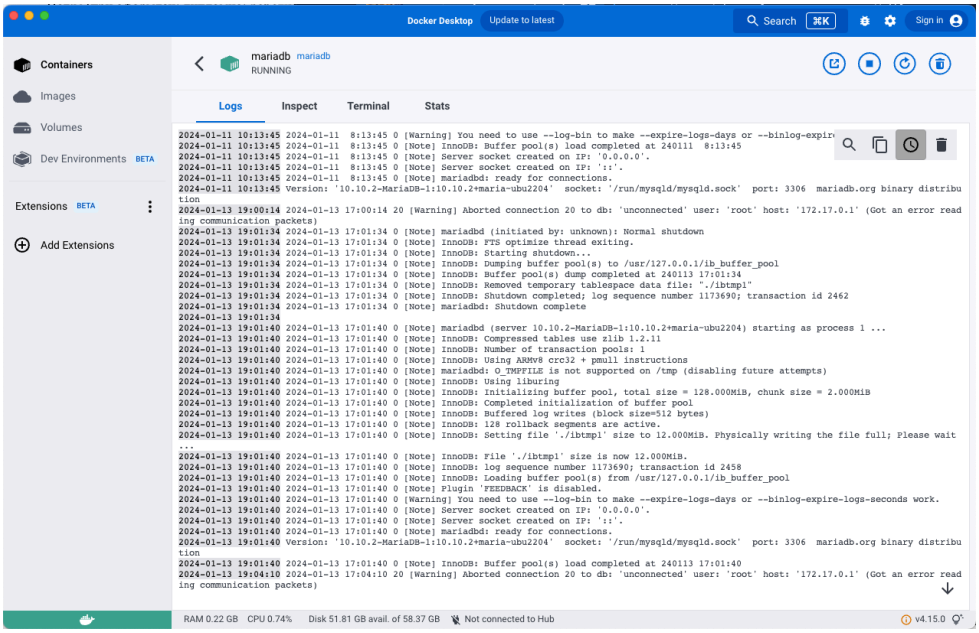


Рисунок 8.1 – Запуск віртуальної машини

Можна подивитись на вікно відображення всіх баз даних, які доступні до запуску на рисунку 8.2.

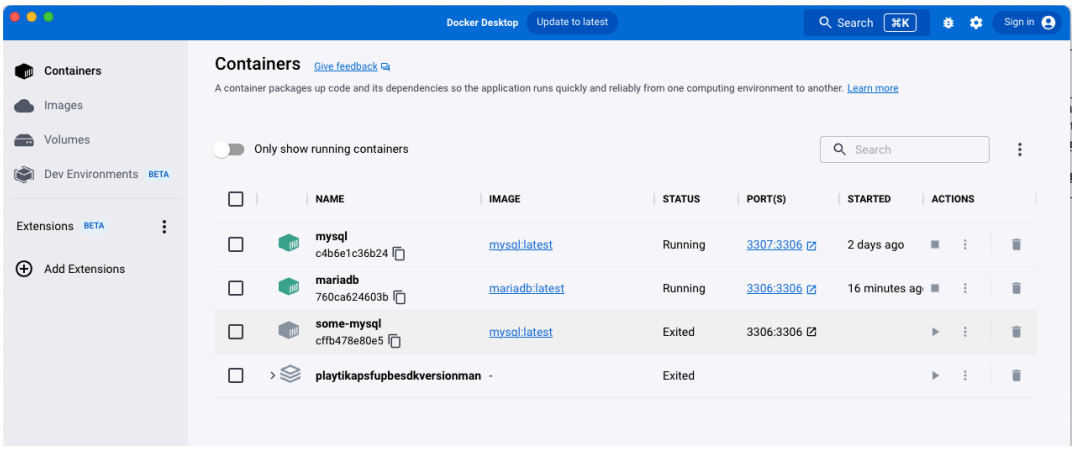


Рисунок 8.2 – Перелік встановлених баз даних

Взаємодія із програмою побудована на послідовному внесенні певних параметрів експериментів.

На першому етапі користувач має можливість обрати одну базу даних для поточного експерименту (рис. 8.2) та параметри, серед запропонованих, на яких необхідно провести експеримент. Вибір функції виконується або з переліку можливих завдяки клавішам навігації на клавіатурі, або введення прямого параметра. В тому випадку, якщо користувач введе неправильний номер функції, програма сповістить його відповідним повідомленням.

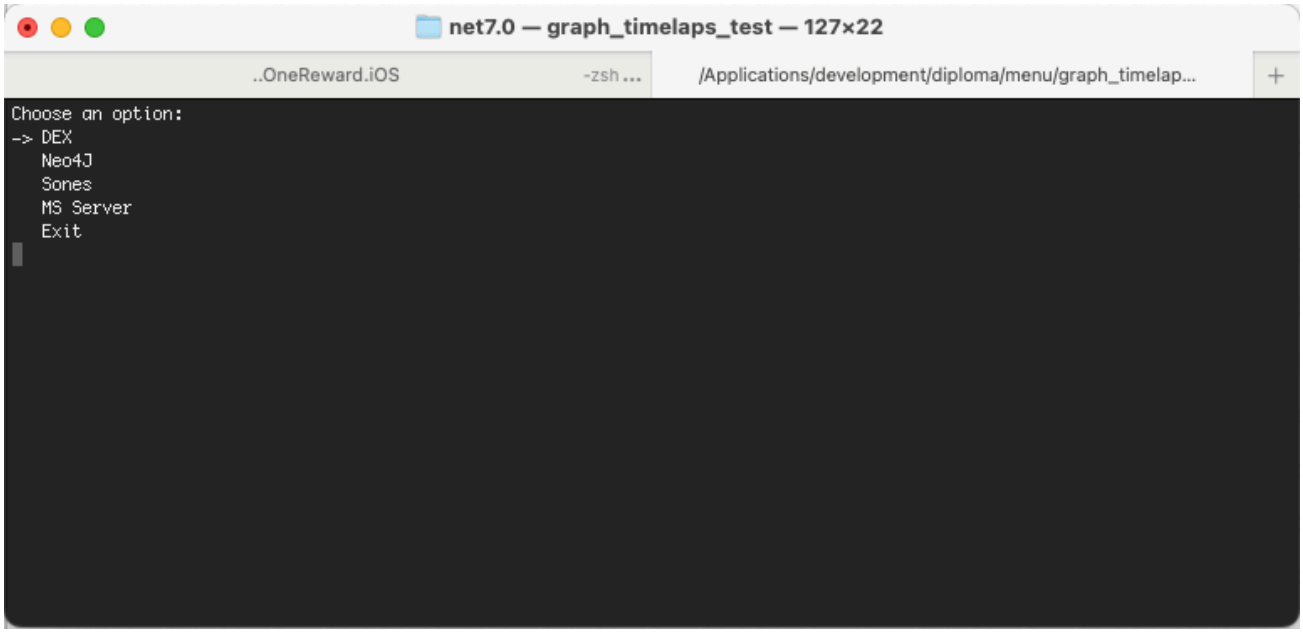


Рисунок 8.2 – Головне меню дослідницької програми

Після вибору певної бази даних буде запропоновано ввести параметри доступу до бази даних (рис. 8.3).

Після успішного встановлення зв'язку між програмою та базою даних, відкриваються можливості для проведення експерименту. Основним кроком є введення параметрів обраного штампу даних, які будуть використовуватися під час експерименту. У нашому прикладі, ці параметри включають глибину сусідів, на яку буде націлено запит, а також максимальну кількість вершин та ребер, які обмежують запит до бази даних. Після цього програма надає можливість обрати тип запиту для тестування (див. рис. 8.4). Цей етап є ключовим для подальших досліджень та аналізу ефективності графової бази даних.

Після вибору типу запиту для експерименту, ми бачимо процес виконання експерименту через бігаючу рисочку прогресу на рисунку 8.5.

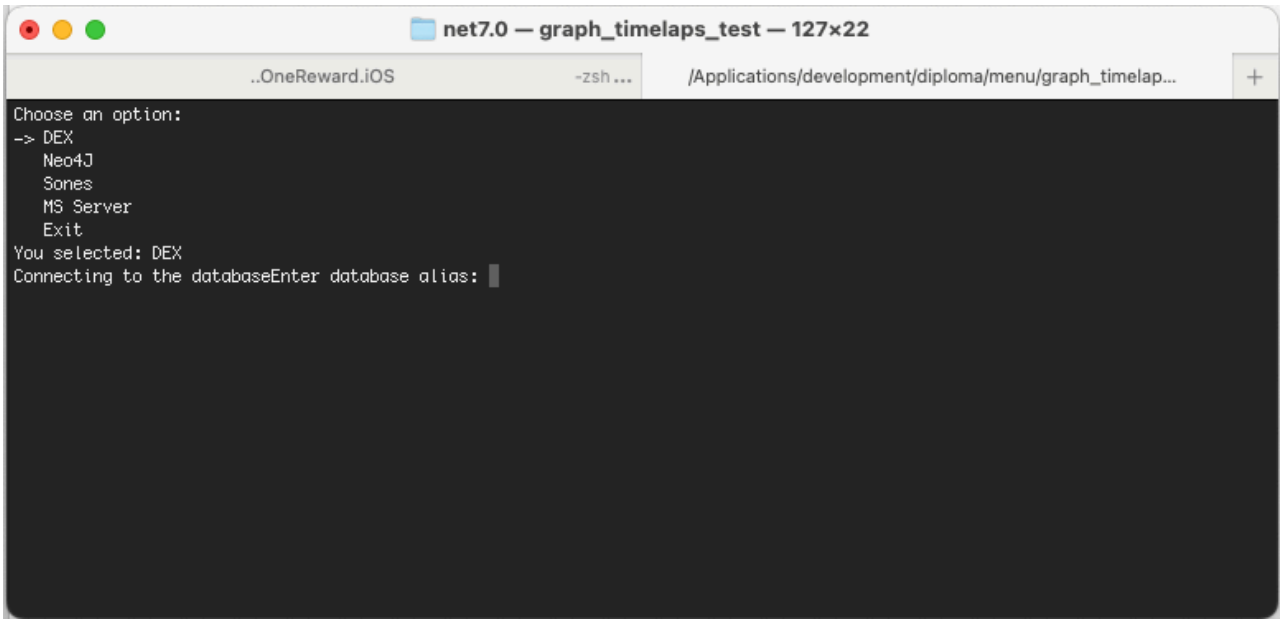


Рисунок 8.3 – Введення параметрів доступу

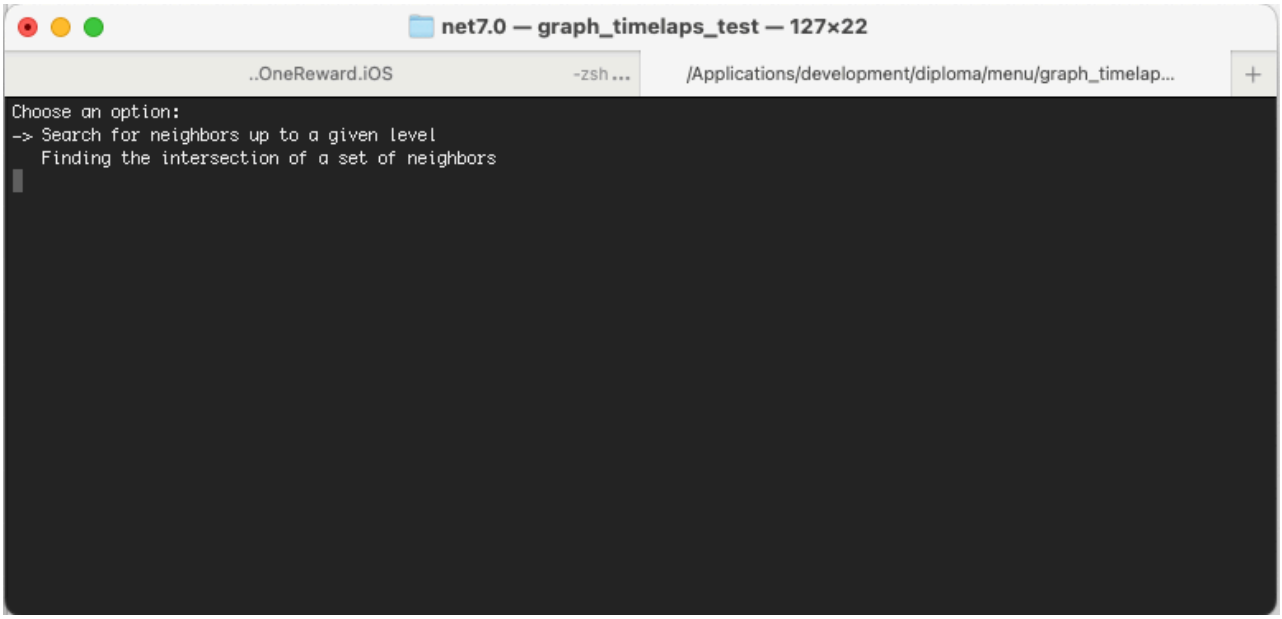


Рисунок 8.4 – Вибір типу запиту для експерименту

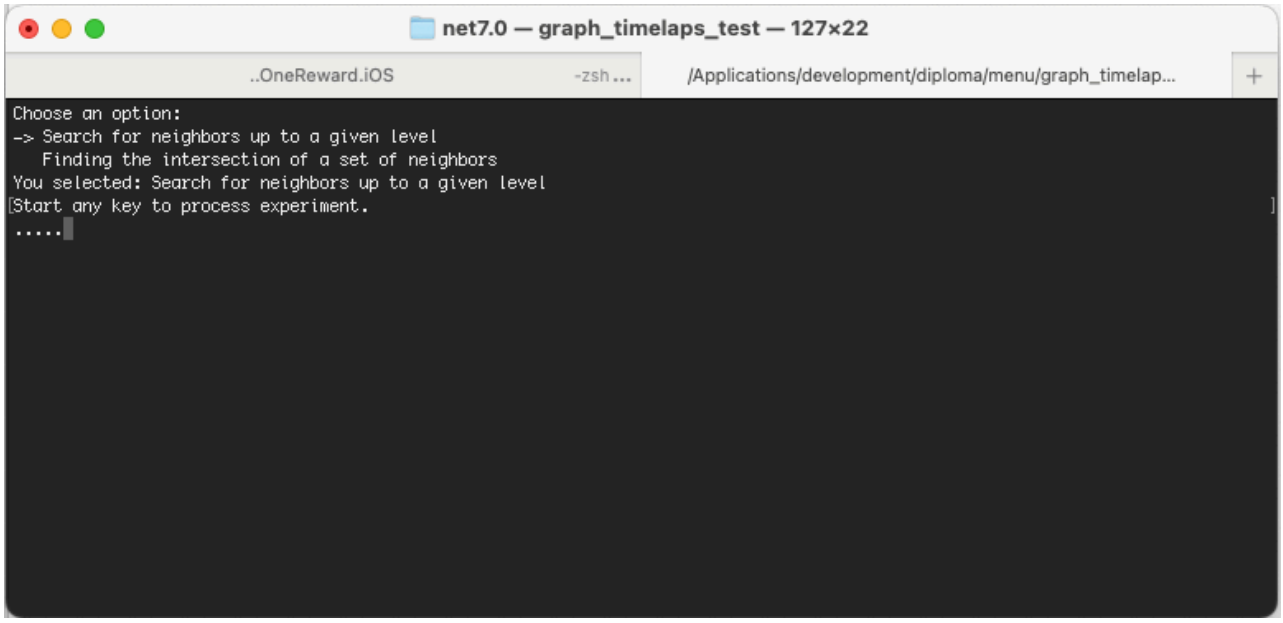


Рисунок 8.5 – Вибір типу запиту для експерименту

Після завершення експерименту його результати будуть збережені у файлі, що дозволяє користувачеві переглядати та витягти отримані дані для подальшого порівняння. Необхідно відзначити, що безпосередньо після завершення експерименту користувач може ініціювати новий експеримент, обравши інший тип запиту. Це удосконалення реалізовано для уникнення повторного введення тих самих даних і сприяє проведенню серії експериментів для отримання об'єктивних результатів дослідження. Кожен експеримент включає мінімум 8 однакових запусків з метою здобуття середнього значення, яке є більш об'єктивним для подальшого аналізу.

Для завершення роботи з програмою, відбувається через закриття консольного вікна або введення комбінації клавіш Ctrl + Z.

9 ПОРЯДОК РОБОТИ З ПРОГРАМОЮ

Порядок роботи з програмою визначено наступним чином:

– розпочати роботу з програмою необхідно із запуску відповідного виконуваного файлу;

– файли розташовані разом із програмою не можна модифікувати чи видаляти - це може призвести до унеможливлення функціонування програмного додатку;

– для перегляду логів програми, в тому числі повідомлень помилок, потрібно відкрити текстовий файл results.txt. Файл логів можна як модифікувати так і видаляти.

10 ПОВІДОМЛЕННЯ

Повідомлення, які отримує користувач від системи наведено в таблиці 10.1.

Таблиця 10.1 – Повідомлення користувачу

Текст повідомлення	Опис ситуації	Рекомендовані дії
Connection Established Successfully	Програма успішно встановила з'єднання з базою даних.	Не потрібно жодних дій. З'єднання встановлено, і програма готова до подальших дій
Failed Connection Attempt	Програма стикнулася з проблемою при спробі підключення до бази даних.	Перевірте доступність мережі, облікові дані бази даних та переконайтеся, що сервер бази даних працює.
Data Read Successfully	Програма успішно прочитала дані з бази даних.	Не потрібно жодних дій. Дані успішно отримані для подальшої обробки.
Failed Data Reading	Програма стикнулася з помилкою при спробі читання даних з бази даних.	Перегляньте деталі помилки. Перевірте коректність запиту на дані та схему бази даних.
Query Execution Successful	Програма успішно виконала запит до бази даних.	Не потрібно жодних дій. Запит виконано, і результати (якщо є) доступні.
Query Execution Error	Під час виконання запиту до бази даних виникла помилка.	Перегляньте деталі помилки. Перевірте синтаксис запиту та його сумісність з базою даних.
Time Measurement	Програма виміряла час виконання конкретного запиту.	Проаналізуйте вимірювання часу для можливостей оптимізації. Розгляньте оптимізацію запиту чи структури бази даних.
Execution Delay Detected	Програма виявила неочікуване затримку під час виконання запиту.	Дослідіть можливі проблеми. Перевірте завантаженість сервера, мережеву затримку чи інші фактори, що впливають на продуктивність.

Закінчення таблиці 10.1

Program Completed Successfully	Програма успішно завершила своє виконання.	Перегляньте загальний час виконання. Помилок не повідомлено; програма завершила свої завдання.
General Optimization Tips	Загальні поради щодо оптимізації запитів та роботи з базою даних.	Розгляньте оптимізацію складних запитів, індексацію таблиць та забезпечення ефективних стратегій вибірки даних.

ЗАТВЕРДЖЕНО
44165850.01360-01 І3 01-ЛЗ

ДОДАТОК Д

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Керівництво користувача

Листів 15

2024

АНОТАЦІЯ

Документ 1116130.01167-01 І3 01 «Програмна система «Database Timelaps Test». Керівництво користувача. Керівництво дослідника» входить до складу програмної документації для програмного інструменту дослідження часової ефективності графових баз даних.

В документі міститься інформація, необхідна для належного використання системи. В документі наведено інформацію про: область застосування, призначення та умови застосування програми, можливості програми, аварійні ситуацій програми та дії при них.

Документ-керівництво також містить контрольний приклад проведення експериментів.

ЗМІСТ

ВСТУП

Область застосування. Програмним додатком «Database Timelaps Test» може користуватись будь-який користувач, який володіє базовими навичками роботи з комп’ютером та консольним вікном, володіє базовими навичками роботи з графовими та реляційними базами даних, розуміння запитів до бази даних, як до реляційних, так і до графових.

Можливості системи. Система надає наступні можливості користувачу:

– дослідження ефективності різних баз даних на прикладі різних вибірок даних;

– вибір різних типів запитів – для об’єктивності порівняння швидкості СУБД при роботі з різними запитами;

– відображати результати роботи програми як на екрані консольного вікна (які включена можливість відображення повної інформації під час роботи програми), так і записувати ці результати у файл.

Рівень підготовки користувача визначається його знаннями та вміннями, необхідними для ефективного використання програмного застосунку та його складових. Користувач повинен ознайомитися з документацією, володіти базовими навичками роботи з графовими та реляційними базами даних, розуміти сутність запитів до обох типів баз даних. Крім того, важливим є вміння користувача взаємодіяти з комп'ютером та консольним вікном.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

1.1 Види діяльності та функції системи

Програмний інструмент «Database Timelaps Test» надає користувачу можливість дослідження та аналізу часової ефективності графових баз даних.

Розроблена система дозволяє визначити доцільність застосування графових баз даних на певних розмірах вибірки пов’язаних між собою, шляхом проведення досліджень на різних ввідних даних.

«Database Timelaps Test» наділений наступними функціональними можливостями:

– вибір набору даних (заповненої бази даних), над якою буде проводитись експеримент;

– введення необхідних параметрів для авторизації в системі управління базами даних (СУБД);

– визначення типу запиту, часову метрику якого слід вимірювати;

– вибір граничного розміру даних, який буде використаний у конкретному експерименті;

– отримання файлу з записами (логуванням) інформації та результатами експерименту;

– можливість вибору типу використовуваної СУБД, контрактного програмного інтерфейсу для взаємодії з обраним API для певної СУБД.

1.2 Умови застосування

Для нормальної роботи програмного застосунку потрібно враховувати такі мінімальні апаратні вимоги:

– пристутність монітора або іншого електронного дисплею;

– наявність маніпулятора типу "миша";

– не менше 5 Мб вільного місця на носії для зберігання даних;

– процесор з тактовою частотою не менше 1 ГГц;

– 8096 Мб оперативної пам'яті.

Програмні вимоги:

– операційна система Windows – програмний додаток створений на платформі Java SE;

– має бути інстальований Java SE (середовище виконання);

– має бути інстальоване Java SE 21.0.1;

– має бути інстальоване Docker Desktop (virtual machine);

– має бути інстальоване MariaDB, MS SQL Server 2016;

– має бути інстальоване DEX Database API;

– має бути інстальоване Sones Database API;

– має бути інстальоване Neo4j Database API + Neo4j Browser

– має бути інстальоване програмне забезпечення для можливості відкриття текстових файлів формату .txt – для перегляду файлів логування роботи програми.

Вхідною інформацією є параметри експерименту:

– імпортовані дані, на яких буде виконуватись експеримент (обрано взаємозв’язки вулиць США та Канади;

– необхідні параметри для авторизації в системі СУБД;

– тип запиту, часова метрика якого буде вимірюватись;

– граничний розмір даних, який повинен бути використаний в конкретному експерименті;

– рівень пошуку сусідніх вершин;

– тип використовуємої СУБД, контрактний програмний інтерфейс для роботою з обраним API для певної СУБД.

2 ПІДГОТОВКА ДО РОБОТИ

2.1 Склад і зміст дистрибутивного носія даних

До складу дистрибутивного носія даних або ж хмарного розміщення програмної системи входить:

– виконуваний файл Database_Timelaps_Test.exe;

– файли необхідних динамічно-приєднуваних бібліотек для функціонування програми;

– електрона версія керівництва користувача-дослідника;

– електрона версія опису програми.

2.2 Порядок загрузки даних і програм

Щоб розпочати використання програмного інструменту, слід виконати такі кроки:

– завантажте всі файли з дистрибутивного носія даних або хмарного сховища;

– знаходячись в папці із файлами програмного додатку, запустіть файл Database_Timelaps_Test.exe;

– відкрийте необхідну базу даних на віртуальній машині та встановіть з'єднання із програмним додатком Database_Timelaps_Test;

– введіть необхідні дані для конфігурації системи, скориставшись вказівками програмного додатку.

2.3 Порядок перевірки працездатності

Для перевірки ефективності системи слід ініціювати запуск програми. При відсутності помилок система виведе повідомлення, яке вказує на потребу обрати конкретну базу даних для подальшого дослідження.

3 ОПИС ОПЕРАЦІЙ

В інструментарії програми процес введення вхідних параметрів ретельно визначений.

Для задання параметрів необхідно виконувати запити системи. Проте, під час взаємодії із базою даних віртуальна машина, на якій виконується СУБД, повинна перебувати в активному стані.

Операції введення інформації виконуються шляхом введення числових або символьних даних і натискання клавіші Enter. Також є можливість вибору варіантів з меню за допомогою клавіш зі стрілками (вгору та вниз).

Вибір здійснюється шляхом введення цілочисельного, невід'ємного значення або символьного значення без коми, наприклад, для глибини та кількості вершин і ребер. При введенні параметрів із використанням дійсних значень, роздільним символом для цілої та дробової частини слід використовувати кому.

3.1 Вибір бази даних

На першому етапі користувач має можливість обрати одну базу даних для поточного експерименту (рис. 3.1) та параметри, серед запропонованих, на яких необхідно провести експеримент. Вибір функції виконується або з переліку можливих завдяки клавішам навігації на клавіатурі, або введення прямого параметра. В тому випадку, якщо користувач введе неправильний номер функції, програма сповістить його відповідним повідомленням.

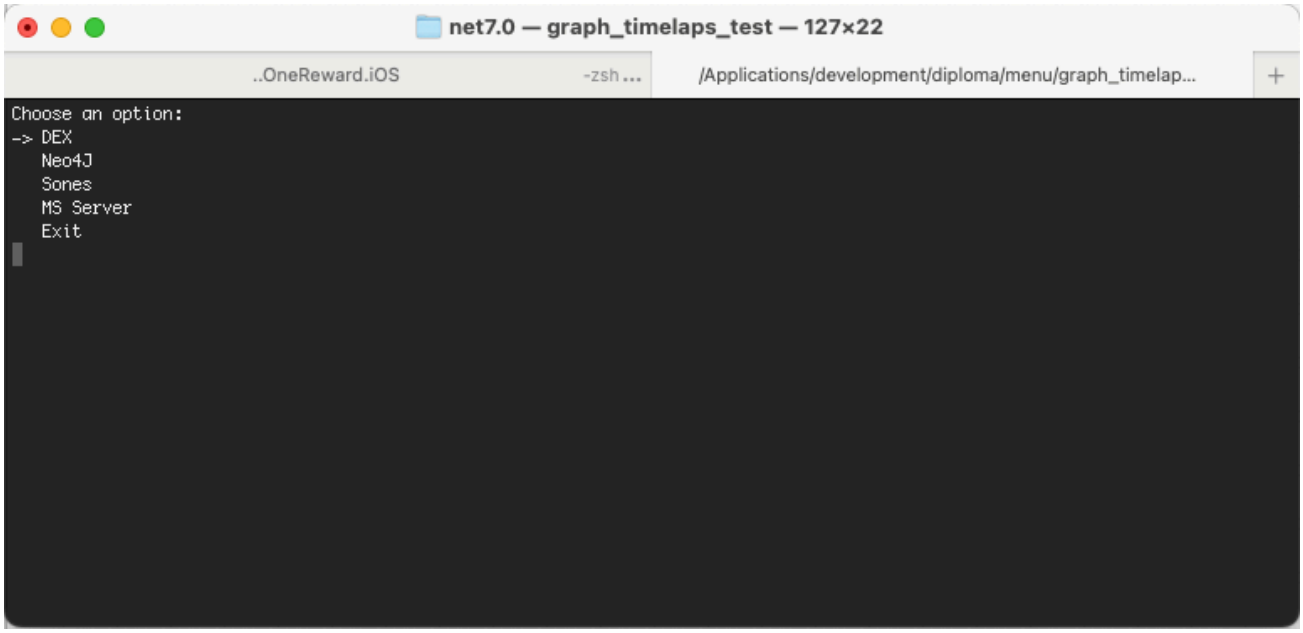


Рисунок 3.1 – Головне меню дослідницької програми

3.2 Введення авторизації для бази даних

Після вибору певної бази даних буде запропоновано ввести параметри доступу до бази даних (рис. 3.2):

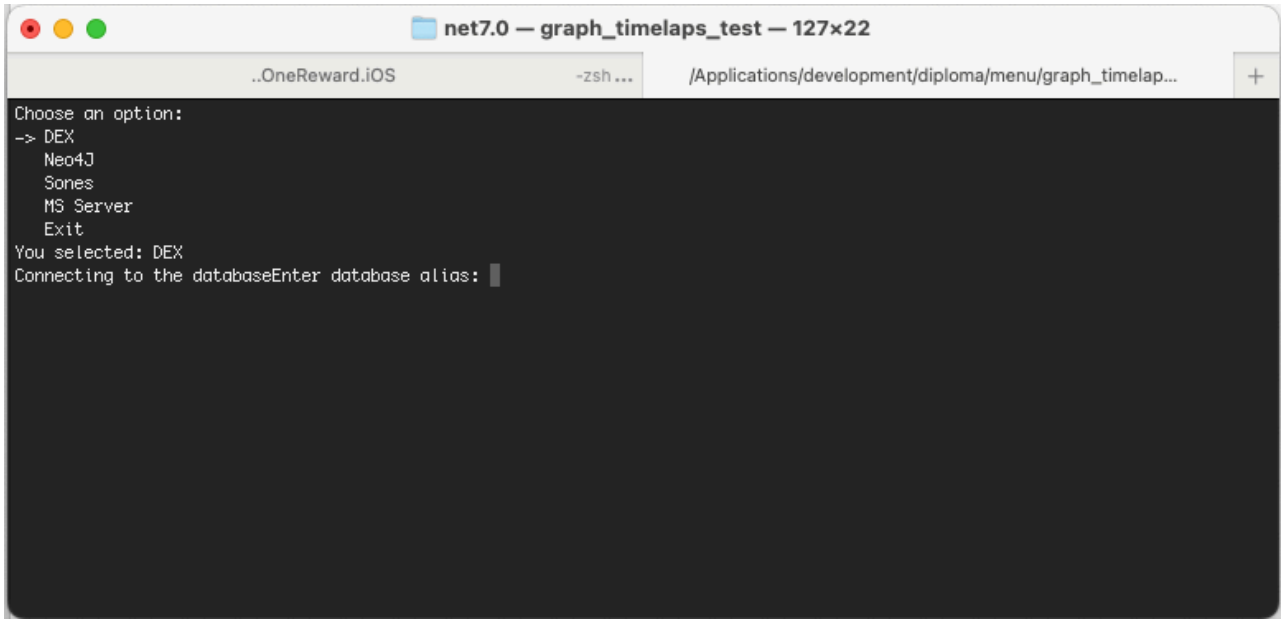


Рисунок 3.2 – Введення параметрів доступу

3.3 Вибір типу запиту для дослідження

Після цього програма надає можливість обрати тип запиту для тестування (див. рис. 3.3). Цей етап є ключовим для подальших досліджень та аналізу ефективності графової бази даних.

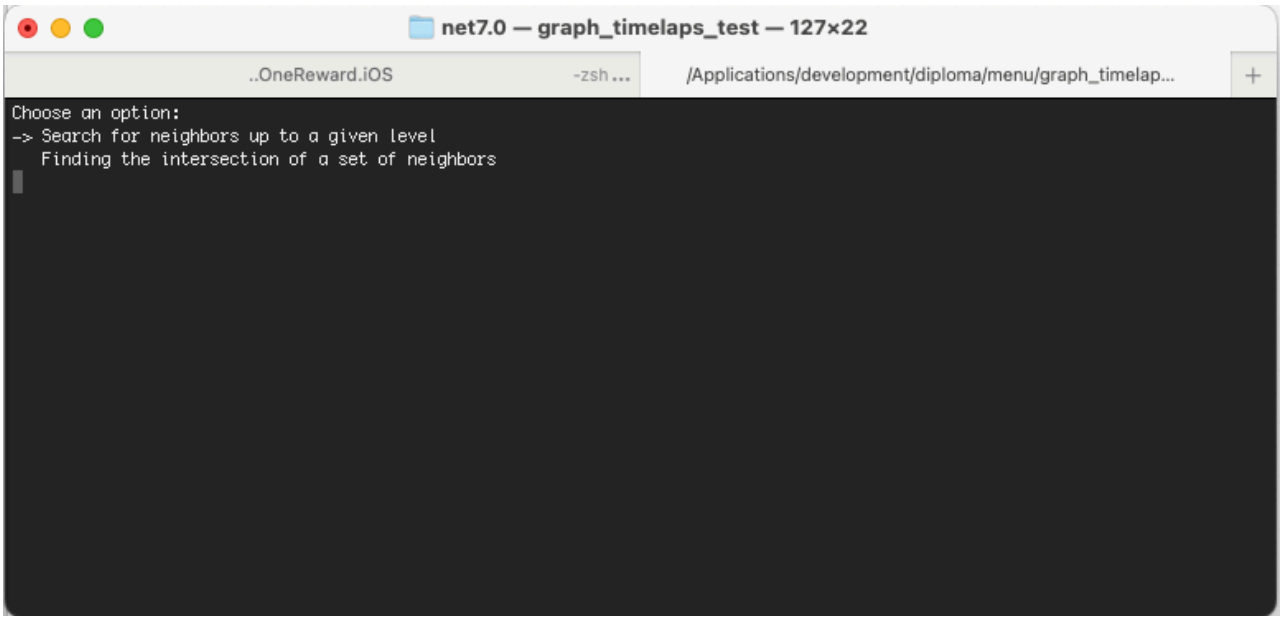


Рисунок 3.3 – Вибір типу запиту для експерименту

3.4 Визначення параметрів експерименту

Після вибору типу запиту для експерименту, ми бачимо процес виконання експерименту через бігаючу рисочку прогресу на рисунку 3.4.

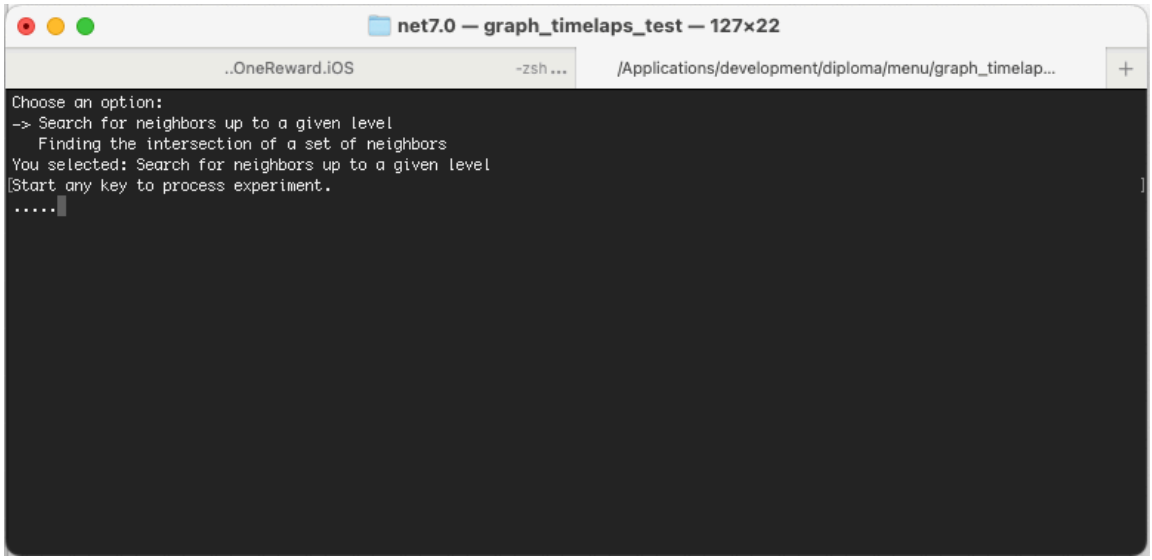


Рисунок 3.4 – Вибір типу запиту для експерименту

Після завершення експерименту його результати будуть збережені у файлі, що дозволяє користувачеві переглядати та витягти отримані дані для подальшого порівняння. Необхідно відзначити, що безпосередньо після завершення експерименту користувач може ініціювати новий експеримент, обравши інший тип запиту.

3.5 Перегляд результатів

Для перегляду результатів отриманих під час роботи програмного додатку – відкрийте текстовий файл «result.txt», який знаходиться в папці «experiments_Results», яка знаходиться в каталозі з виконуваним файлом.

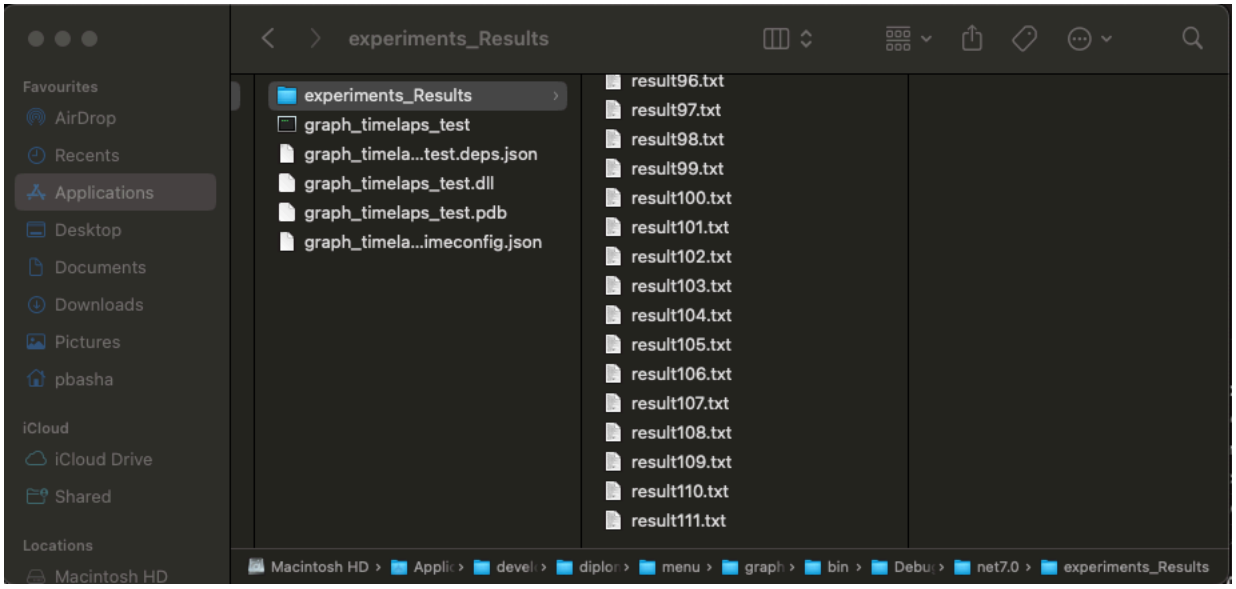


Рисунок 3.5 – Папка з результатами експериментів

Вміст результуючого файлу зображено на рисунку 3.6.

result3.txt

```
Experiment No 3.
Chosen database: neo4j
Chosen request type: Search for neighbors up to a given level
Chosen request depth: 10
Chosen vertices count: 10000
Chosen edges count: 90000
------------------------------
[Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.31-1.el8 started.
[Note] [Entrypoint]: Switching to dedicated user 'mysql'
[Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.31-1.el8 started.
[Note] [Entrypoint]: Initializing database files
[Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future
release. Please use SET GLOBAL host_cache_size=0 instead.
 [System] [MY-013169] [Server] /usr/sbin/mysqld (mysqld 8.0.31) initializing of server in progress as process 81
 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
[System] [MY-013577] [InnoDB] InnoDB initialization has ended.
[Warning] [MY-010453] [Server] root@localhost is created with an empty password ! Please consider switching off
the --initialize-insecure option.
 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future
release. Please use SET GLOBAL host_cache_size=0 instead.
 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.31) starting as process 132
 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now
supported for this channel.
 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is
accessible to all OS users. Consider choosing a different directory.
 [System] [MY-011323] [Server] X Plugin ready for connections. Socket: /var/run/mysqld/mysqlx.sock
 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.31'  socket: '/var/run/
mysqld/mysqld.sock'  port: 0  MySQL Community Server - GPL.
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leapseconds' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/tzdata.zi' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.
[System] [MY-013172] [Server] Received SHUTDOWN from user root. Shutting down mysqld (Version: 8.0.31).
 [System] [MY-010910] [Server] /usr/sbin/mysqld: Shutdown complete (mysqld 8.0.31)  MySQL Community Server - GPL.
 [Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future
release. Please use SET GLOBAL host_cache_size=0 instead.
 [System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.31) starting as process 1
 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
 [Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now
supported for this channel.
 [Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is
accessible to all OS users. Consider choosing a different directory.
 [System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/run/
mysqld/mysqlx.sock
 [System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.31'  socket: '/var/run/
mysqld/mysqld.sock'  port: 3306  MySQL Community Server - GPL.
[Warning] [MY-011068] [Server] The syntax '--skip-host-cache' is deprecated and will be removed in a future
release. Please use SET GLOBAL host_cache_size=0 instead.
[System] [MY-010116] [Server] /usr/sbin/mysqld (mysqld 8.0.31) starting as process 1
[Note] [Entrypoint]: Database files initialized
[Note] [Entrypoint]: Starting temporary server
[Note] [Entrypoint]: Temporary server started.
'/var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
[Note] [Entrypoint]: Stopping temporary server
[Note] [Entrypoint]: Temporary server stopped
[Note] [Entrypoint]: MySQL init process done. Ready for start up.
[Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.31-1.el8 started.
[Note] [Entrypoint]: Switching to dedicated user 'mysql'
[Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.31-1.el8 started.
'/var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
[System] [MY-013576] [InnoDB] InnoDB initialization has started.
[System] [MY-013577] [InnoDB] InnoDB initialization has ended.
[System] [MY-010229] [Server] Starting XA crash recovery...
[System] [MY-010232] [Server] XA crash recovery finished.
[Warning] [MY-010068] [Server] CA certificate ca.pem is self signed.
[System] [MY-013602] [Server] Channel mysql_main configured to support TLS. Encrypted connections are now
supported for this channel.
[Warning] [MY-011810] [Server] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is
accessible to all OS users. Consider choosing a different directory.
[System] [MY-011323] [Server] X Plugin ready for connections. Bind-address: '::' port: 33060, socket: /var/run/
mysqld/mysqlx.sock
[System] [MY-010931] [Server] /usr/sbin/mysqld: ready for connections. Version: '8.0.31'  socket: '/var/run/
mysqld/mysqld.sock'  port: 3306  MySQL Community Server - GPL.
------------------------------
Time elapsed for loading: 5.12439 sec
Time elapsed for request: 21.04312 sec
```

Рисунок 3.6 – Файл з результатом експерименту

4 АВАРІЙНІ СИТУАЦІЇ

Однією з основних аварійних ситуацій може бути відмова в роботі програмного додатку або виникнення помилки під час його функціонування. Випадок відмови програмного додатку може виникнути у разі відсутності одного з необхідних бібліотечних файлів.

В такому випадку користувачу рекомендується перевірити наявність всіх файлів, які повинні бути на дистрибутивному носії даних або в хмарному сховищі. Якщо користувач виявить відсутність певних файлів, необхідно провести перевстановлення програмного інструментарію.

Додатково, можлива помилка під час з’єднання з обраною базою даних. Це може статися, якщо програма не може знайти необхідний екземпляр бази даних за обраним портом.

У цьому випадку важливо переконатися, що віртуальна машина запущена, екземпляр бази даних працює в активному режимі, і відсутні будь-які проблеми зі статусом бази даних.

5 РЕКОМЕНДАЦІЇ ЩОДО ЗАСВОЄННЯ

Основною рекомендацією щодо засвоєння та експлуатації є ознайомлення з керівництвом користувача та описом програми, а також дотримання вказівок системи.

Нижче зображено контрольний приклад виконання програми:

Запуск програми:

– вибір бажаної бази даних, з якою буде встановлено зв’язок для дослідження часу запиту – вибір «DEX»;

– введення необхідних даних для авторизації за обраною базою даних: назва бази даних або її alias та пароль доступу до бази даних – «database1» та «admin»;

– введення глибини пошуку сусідів, наприклад – «5»;

– введення максимальної кількості вершин – «5000»;

– введення максимальної кількості ребер – «10000»;

– далі користувачу залишається лише очікувати результати роботи програми та знайти відповідний файл з номером експерименту, наприклад «result37.txt» у папці «experiments_Results», яка знаходиться в каталозі з виконуваним файлом.

6 ПОВІДОМЛЕННЯ

Повідомлення, які отримує користувач від системи наведено в таблиці 6.1.

Таблиця 6.1 – Повідомлення користувачу

Текст повідомлення	Опис ситуації	Рекомендовані дії
Connection Established Successfully	Програма успішно встановила з'єднання з базою даних.	Не потрібно жодних дій. З'єднання встановлено, і програма готова до подальших дій
Failed Connection Attempt	Програма стикнулася з проблемою при спробі підключення до бази даних.	Перевірте доступність мережі, облікові дані бази даних та переконайтеся, що сервер бази даних працює.
Data Read Successfully	Програма успішно прочитала дані з бази даних.	Не потрібно жодних дій. Дані успішно отримані для подальшої обробки.
Failed Data Reading	Програма стикнулася з помилкою при спробі читання даних з бази даних.	Перегляньте деталі помилки. Перевірте коректність запиту на дані та схему бази даних.
Query Execution Successful	Програма успішно виконала запит до бази даних.	Не потрібно жодних дій. Запит виконано, і результати (якщо є) доступні.
Query Execution Error	Під час виконання запиту до бази даних виникла помилка.	Перегляньте деталі помилки. Перевірте синтаксис запиту та його сумісність з базою даних.
Time Measurement	Програма виміряла час виконання конкретного запиту.	Проаналізуйте вимірювання часу для можливостей оптимізації. Розгляньте оптимізацію запиту чи структури бази даних.
Execution Delay Detected	Програма виявила неочікуване затримку під час виконання запиту.	Дослідіть можливі проблеми. Перевірте завантаженість сервера, мережеву затримку чи інші фактори, що впливають на продуктивність.

Закінчення таблиці 6.1

Текст повідомлення	Опис ситуації	Рекомендовані дії
Program Completed Successfully	Програма успішно завершила своє виконання.	Перегляньте загальний час виконання. Помилок не повідомлено; програма завершила свої завдання.
General Optimization Tips	Загальні поради щодо оптимізації запитів та роботи з базою даних.	Розгляньте оптимізацію складних запитів, індексацію таблиць та забезпечення ефективних стратегій вибірки даних.

ЗАТВЕРДЖЕНО
44165850.01360-01 12 01

ДОДАТОК Е

Технічне Завдання

ПРОГРАМНА СИСТЕМА DATABASE TIMELAPS TEST

Текст програми
Листів 25

2024

АНОТАЦІЯ

Документ 44165850.01360-01 12 01 «Програмна система Database Timelaps Test. Текст програми». входить до складу програмної документації для програмного інструменту дослідження часової ефективності графових баз даних у порівнянні з реляційними на різних об’ємах пов’язаних.

В даному документі представлено текст програмного інструменту. Програму було розроблено в середовищі розробки IntelliJ IDEA Community із застосуванням мови програмування Java.

ЗМІСТ

1 СТРУКТУРА ПРОГРАМИ

1.1 Використанні залежності

Для розробленої системи існують наступні залежності:

– NLog.config – файл конфігурації логів системи роботи програмної системи;

– також існують залежності на контракти роботи з різними базами даних, задля того, щоб мати змогу розробляти програмний продукт для дослідництва певних баз даних, тож використані наступні пакети: Neo4J api driver, DEX api driver, Sones api driver.

Під час створення архітектури системи було отримано наступну діаграму класів та інтерфейсів (рис. 1.1). Слід урахувати що на діаграмі зображено абстрактний шар загальної системи необхідної для її повноцінного функціонування.

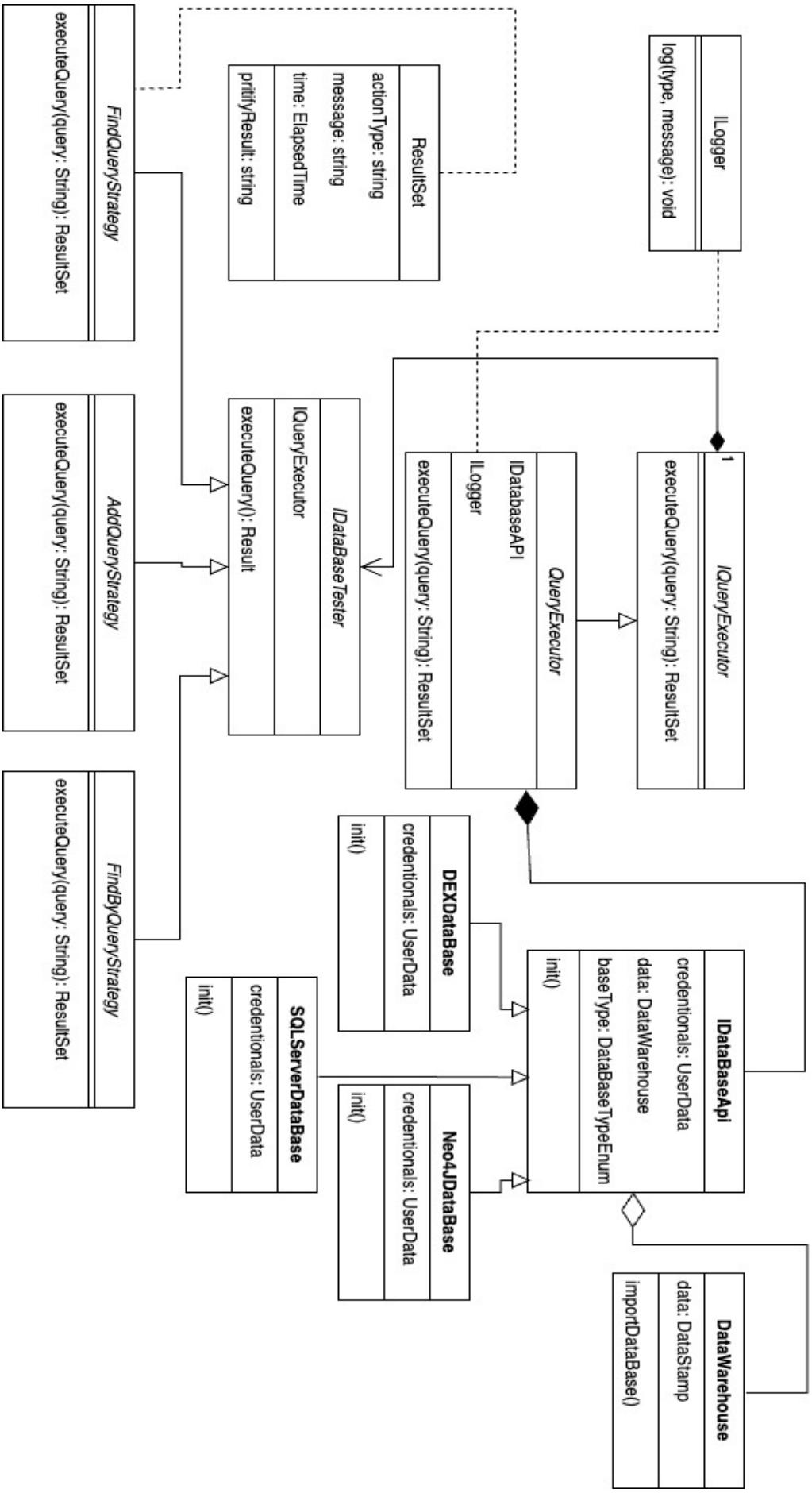


Рисунок 1.1 – UML схема взаємодії основних елементів системи

За рахунок використання поліморфізму було отримано наступні ієрархічні зв’язки:

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні реалізовувати інтерфейс IDatabaseApi;

– усі контракти різних типів баз даних, які будуть використовуватись в дослідженні повинні успадковувати клас BaseEntryDatabase;

– усі варіанти логування повинні реалізовувати інтерфейс ILogger;

– інтерфейс IDataBaseTaster повинен використовувати тільки реалізації інтерфейсу IDatabaseApi для відтворення запитів до бази даних.

Ці відносини забезпечують стабільність та можливість розширення системи, дозволяючи легко додавати нові компоненти чи змінювати існуючі, при цьому забезпечуючи відповідність до визначених інтерфейсів.

Таблиця 1.1 – Основні класи та інтерфейси

Назва елементу	Опис
DatabaseTester	Відповідає за тестування часової ефективності графових та реляційних баз даних.
IDatabaseApi	Інтерфейс який дає контракт використання будь якої бази даних. Послідовники GraphDatabase та RelationalDatabase визначають типи баз даних, з якими система взаємодіє. GraphDatabase відповідає за графові бази даних, тоді як RelationalDatabase представляє реляційні бази. Ці класи надають необхідний інтерфейс для виконання запитів та управління даними відповідного типу.
QueryExecutor	Відіграє роль виконавця запитів до баз даних. Він отримує SQL- або графові запити від інших частин системи та відповідає за їхнє виконання, повертаючи результати у вигляді ResultSet.

Продовження таблиці 1.1

Назва елементу	Опис
ResultSet	Служить для представлення результатів виконаних запитів до бази даних. Його завдання - забезпечити структуроване представлення даних, отриманих в результаті виконання запитів.
ILogger	Визначає загальний інтерфейс для логування подій та даних в системі. Цей інтерфейс дозволяє різним класам системи взаємодіяти із системою логування.
FileLogger та ConsoleLogger	Реалізують інтерфейс ILogger і відповідають за логування інформації у файлі та консолі відповідно. Ці класи забезпечують різні методи виведення інформації.
ExperimentResult	Клас, що представляє результати конкретного експерименту. Він містить інформацію про часові показники, статистику та інші деталі, які можуть бути важливими для подальшого аналізу.
DataWarehouse	Клас визначає сховище даних, де зберігаються результати експериментів. Цей клас дозволяє управляти та зберігати експериментальні дані для подальшого використання та аналізу.
DataStamp	Клас відповідає за створення або імпорт штампу даних, який використовується для налаштування експерименту.
Node, Edge, Table, Column, DataType	Визначають основні елементи, що використовуються для представлення структури даних у графових та реляційних базах даних.
GraphData та RelationalData	Визначають структуру та особливості графових та реляційних даних відповідно. Вони представляють об'єкти, з якими буде взаємодіяти система.
DataBaseTransformator	Класо, який відповідає за трансформацію даних. Він може забезпечити конвертацію даних з одного формату до іншого для оптимізації експерименту.

1.2 Принципи та шаблони проектування

Під час проектування внутрішньої структури системи використовувався принцип SOLID [23]. Даний принцип надає можливість створювати системи, які можна буде легко підтримувати і розширювати функціонал у майбутньому.

При розробці архітектури програми для визначення часової ефективності графових баз даних можна використовувати ряд принципів та шаблонів проєктування. Нижче наведено деякі з них:

– принцип єдиної відповідальності (Single Responsibility Principle): клас DatabaseTester вже реалізує функціонал тестування баз даних, QueryExecutor - виконання запитів, ILogger - логування. Кожен клас має чітко визначену відповідальність, що забезпечує чистоту та прозорість коду;

– принцип відкритості/закритості (Open/Closed Principle): система побудована з урахуванням можливості розширення новими класами для різних типів баз даних. Код вже закритий для змін, але відкритий для розширень;

– принцип заміщення Лісков (Liskov Substitution Principle): усі класи баз даних вже реалізують інтерфейси GraphDatabase та RelationalDatabase. Об'єкти базового класу можна замінювати його похідними без порушення коректності системи;

– шаблон Стратегія (Strategy): клас DatabaseTester вже містить механізм вибору стратегії тестування, де різні стратегії реалізуються як окремі об'єкти (рис. 1.3);

– шаблон Декоратор (Decorator): застосування декоратора для функціональності логування реалізовано через класи FileLogger, ConsoleLogger та інші, які додають можливість логування до основного класу (рис. 1.4);

– шаблон Фабричний метод (Factory Method): система вже використовує фабричний метод для створення об'єктів різних баз даних без прив'язки до конкретних класів (рис. 1.5);

– шаблон Оголошення процедури (Observer): клас ExperimentResult вже є спостерігачем, який отримує та обробляє повідомлення про результати експериментів від інших частин системи (рис. 1.6);

– шаблон Ланцюг відповідальності (Chain of Responsibility): система вже містить ланцюг об'єктів логування, які обробляють логи та передають їх далі у ланцюгу;

– шаблон Фасад (Facade): застосування фасаду реалізовано для об'єднання складних підсистем, таких як логування, та надання простого інтерфейсу для взаємодії;

– шаблон Знімок (Memento): система вже використовує шаблон "знімок" для збереження та відновлення стану експериментів.

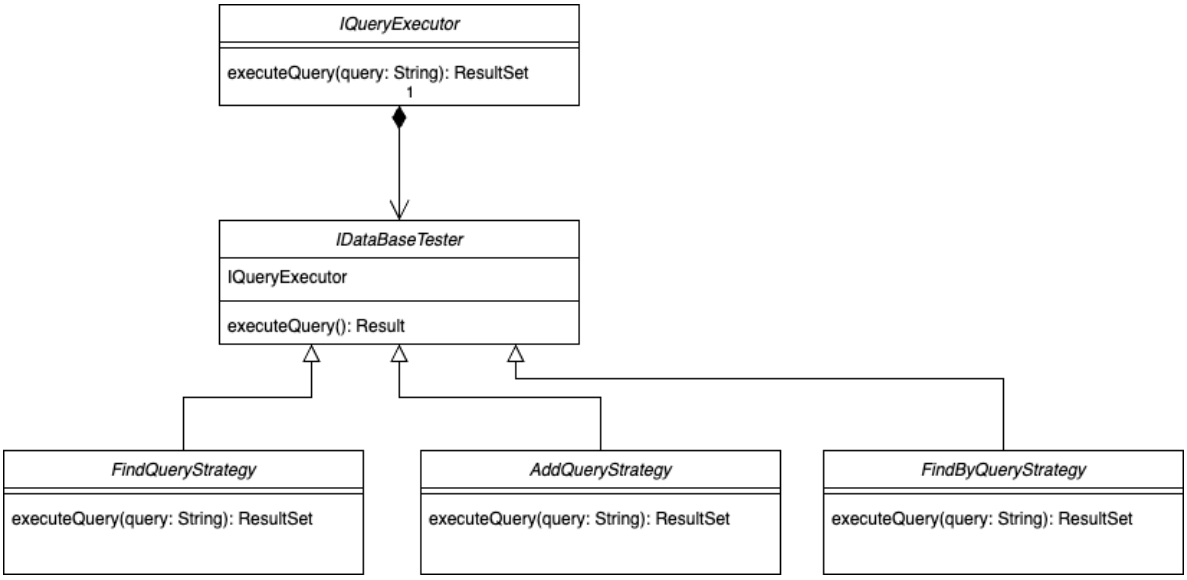


Рисунок 1.3 – Приклад використання спрощеної версії шаблону «абстрактна фабрика»

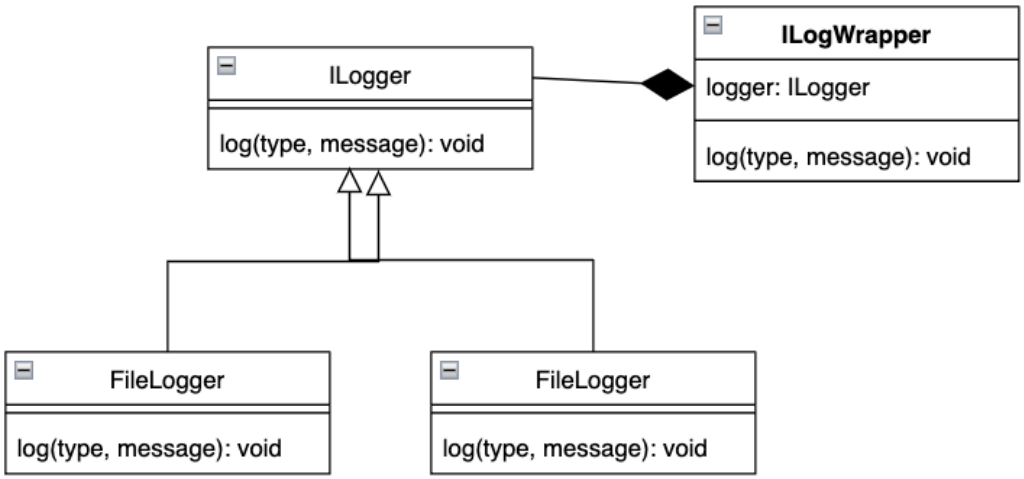


Рисунок 1.4 – Приклад застосування шаблону Decorator

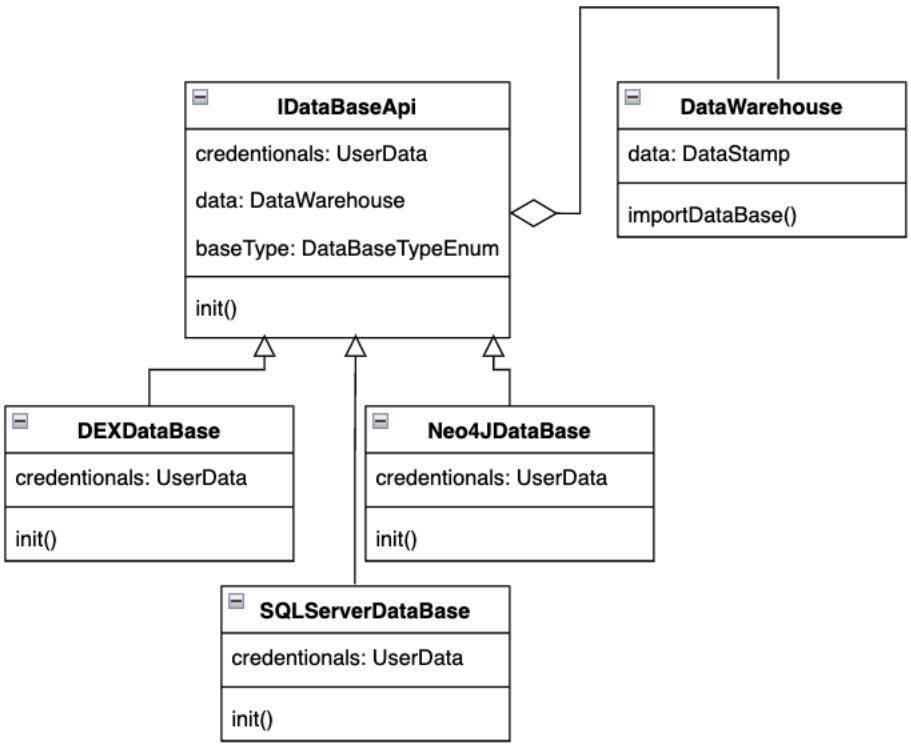


Рисунок 1.5 – Приклад застосування шаблону «Абстрактний метод»

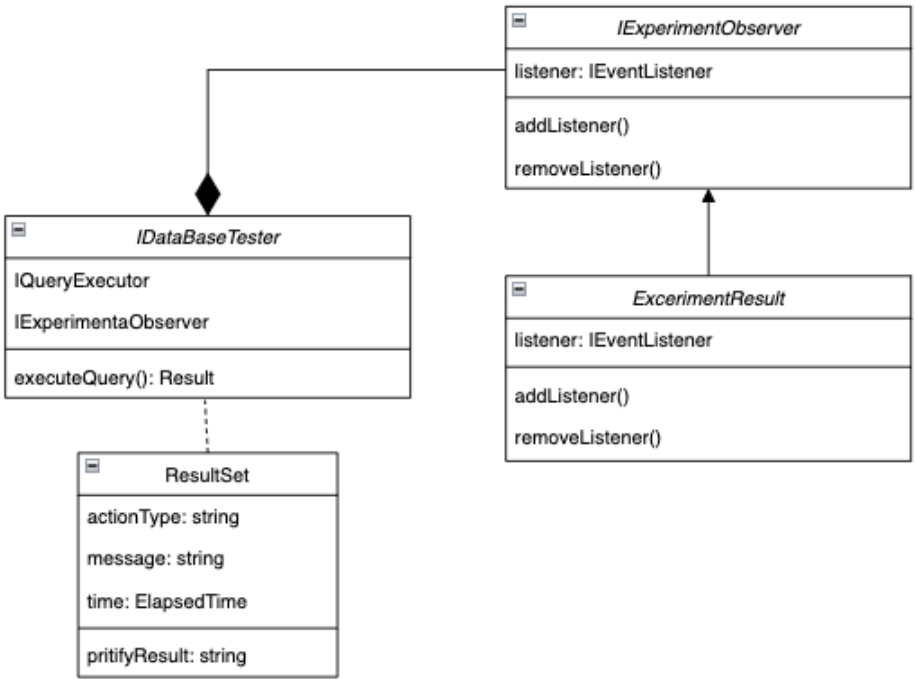


Рисунок 1.6 – Приклад застосування шаблону «Observer»

2 ТЕКСТ ПРОГРАМИ

2.1 Текст класу Main.java

```
public class Main {

    public static void main(String[] args) {
        try {
            runApplication();
            Logger.close();
        } catch (Exception e) {
            Logger.close();
        }
    }

    private static void runApplication() {
        startMenu();

        HashMap<String, String[]> sqlDatabases
= initializeSqlDatabases();
        HashMap<String, String> neo4jSettings
= initializeNeo4jSettings();

        DataGenerator dataGenerator = new
DataGenerator(sqlDatabases, neo4jSettings,
dbMariadbUrl);

executeDataGenerationTests(dataGenerator);

        QueryTester queryTester = new
QueryTester(sqlDatabases, neo4jSettings);

        Logger.log("\nNO INDEXES\n");

        Boolean showAll = false;

        executeQueryTests(queryTester,
showAll);
        executeIndexTests(dataGenerator,
queryTester, showAll);
        executeDeletionTests(dataGenerator,
queryTester, showAll);

        Logger.log("\nREMOVING MySQL\n");

sqlDatabases.remove("jdbc:mysql://127.0.0.1:33
07/");

        executeQueryTests(queryTester,
showAll);
        executeIndexTests(dataGenerator,
queryTester, showAll);
        executeDeletionTests(dataGenerator,
queryTester, showAll);

        HashMap<String, Integer>
customerInvoice =
dataGenerator.insertSequentialInvoices(1, 10,
100);

        int invoiceIndex =
customerInvoice.get("invoiceIndex");
        int customerIndex =
customerInvoice.get("customerIndex");


executeRecursiveQueryTests(queryTester,
showAll, invoiceIndex);

        Logger.log("customerIndex " +
customerIndex);

dataGenerator.cleanSequentialInvoices(customer
Index);

        customerInvoice =
dataGenerator.insertSequentialInvoices(1, 10,
1000);

        invoiceIndex =
customerInvoice.get("invoiceIndex");


executeRecursiveQueryTests(queryTester,
showAll, invoiceIndex);


dataGenerator.cleanSequentialInvoices(customer
Index);

        executeIndexTests(dataGenerator,
queryTester, showAll);
        executeDeletionTests(dataGenerator,
queryTester, showAll);

        customerInvoice =
dataGenerator.insertSequentialInvoices(1, 10,
100);

        invoiceIndex =
customerInvoice.get("invoiceIndex");
        customerIndex =
customerInvoice.get("customerIndex");


executeRecursiveQueryTests(queryTester,
showAll, invoiceIndex);

        Logger.log("customerIndex " +
customerIndex);

dataGenerator.cleanSequentialInvoices(customer
Index);

        customerInvoice =
dataGenerator.insertSequentialInvoices(1, 10,
1000);

        invoiceIndex =
customerInvoice.get("invoiceIndex");


executeRecursiveQueryTests(queryTester,
showAll, invoiceIndex);


dataGenerator.cleanSequentialInvoices(customer
Index);
    }

    private static void
executeDataGenerationTests(DataGenerator
dataGenerator) {
        dataGenerator.createTables();

dataGenerator.createSampleTables(dbMariadbUrl)
;
        dataGenerator.loadSampleData(10,
dbMariadbUrl);
```

```
dataGenerator.insertItemsAndWorkTypes(10, 10,
10000, 10000);
        dataGenerator.insertWorkData(10, 1000,
10, 10, 10);
        dataGenerator.insertCustomerData(10,
1000, 10, 10, 0, 10, 10);
    }

    private static void
executeQueryTests(QueryTester queryTester,
boolean showAll) {
        queryTester.executeQueryTestsSQL(12,
showAll);

queryTester.executeQueryTestsCypher(12,
showAll);
    }

    private static void
executeIndexTests(DataGenerator dataGenerator,
QueryTester queryTester, boolean showAll) {
        Logger.log("\nCREATING INDEXES\n");
        dataGenerator.createIndexes();
        executeQueryTests(queryTester,
showAll);
    }

    private static void
executeDeletionTests(DataGenerator
dataGenerator, QueryTester queryTester,
boolean showAll) {
        Logger.log("\nDELETING INDEXES\n");
        dataGenerator.deleteIndexes();

executeComplexQueryTests(dataGenerator,
queryTester, showAll);
    }

    private static void
executeRecursiveQueryTests(QueryTester
queryTester, boolean showAll, int
invoiceIndex) {

queryTester.executeRecursiveQueryTestSQL(12,
showAll, invoiceIndex);

queryTester.executeRecursiveQueryTestCypher(12
, showAll, invoiceIndex);
    }

    private static void startMenu() {
        simulateDatabaseConnection();

        Scanner scanner = new
Scanner(System.in);
        int choice;
        do {
            clearConsole();
            printMenu();
            choice = scanner.nextInt();
            processUserChoice(choice);
        } while (choice != 4);
    }

    private static void printMenu() {
        clearConsole();
        System.out.println("1. Enter database
type");
        System.out.println("2. Enter sampling
depth");
        System.out.println("3. Enter query
type");
        System.out.println("4. Exit");
        System.out.print("Enter your choice:
");
    }

    private static void clearConsole() {
        for (int i = 0; i < 50; i++) {
            System.out.println(); // Print 50
new lines
        }
    }

    private static void processUserChoice(int
choice) {
        Scanner scanner = new
Scanner(System.in);
        switch (choice) {
            case 1:
                System.out.print("Enter
database type: ");
                String databaseType =
scanner.nextLine();
                // Add your code to handle the
entered database type
                break;
            case 2:
                System.out.print("Enter
sampling depth: ");
                int samplingDepth =
scanner.nextInt();
                // Add your code to handle the
entered sampling depth
                break;
            case 3:
                System.out.print("Enter query
type: ");
                String queryType =
scanner.nextLine();
                // Add your code to handle the
entered query type
                break;
            case 4:
                System.out.println("Exiting
the program. Goodbye!");
                break;
            default:
                System.out.println("Invalid
choice. Please enter a valid option.");
        }
    }
}
```

2.2 Текст класу DataObserver.java

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;

import java.sql.*;
import java.util.HashMap;
import java.util.List;

public class DataGenerator {

    private final HashMap<String, String[]>
sqlDatabases;
    private final HashMap<String, String>
neo4jSettings;
```

```
    private final String defaultDbUrl;

    private final List<String> firstnames;
    private final List<String> surnames;
    private final List<HashMap<String,
String>> addresses;

    public DataGenerator(HashMap<String,
String[]> sqlDatabases, HashMap<String,
String> neo4jSettings, String defaultDbUrl) {
        this.sqlDatabases = sqlDatabases;
        this.neo4jSettings = neo4jSettings;
        this.defaultDbUrl = defaultDbUrl;
    }

    public void executeSQLUpdate(String
sqlQuery, String dbUrl, String[] dbSettings) {
        try (Connection conn =
createConnection(dbUrl, dbSettings);
             Statement stmt =
conn.createStatement()) {

            stmt.executeUpdate(sqlQuery);

        } catch (SQLException e) {
            Logger.log("SQLException");
            e.printStackTrace();
        } catch (Exception e) {
            Logger.log("Exception");
            e.printStackTrace();
        }
    }

    public ResultSet executeSQLQuery(String
sqlQuery) {
        try (Connection conn =
createConnection(defaultDbUrl,
sqlDatabases.get(defaultDbUrl));
             Statement stmt =
conn.createStatement()) {

            return
stmt.executeQuery(sqlQuery);

        } catch (SQLException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }

        return null;
    }

    public void truncateDatabases() {
        truncateNeo4jDatabase();
        truncateSqlDatabases();
    }

    public void truncateDatabasesWork() {
        truncateNeo4jDatabaseWork();
        truncateSqlDatabasesWork();
    }

    public void truncateDatabasesCustomer() {
        truncateNeo4jDatabaseCustomer();
        truncateSqlDatabasesCustomer();
    }

    private Connection createConnection(String
dbUrl, String[] dbSettings) throws
SQLException, ClassNotFoundException {
        Class.forName(dbSettings[0]);
        return
DriverManager.getConnection(dbUrl,
dbSettings[1], dbSettings[2]);
    }

    private void truncateNeo4jDatabase() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        try (org.neo4j.driver.Driver driver =
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));
             Session session =
driver.session()) {

            session.run("MATCH (n) DETACH
DELETE n");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void truncateSqlDatabases() {
        for (String dbUrl :
sqlDatabases.keySet()) {
            executeTruncateBatch(dbUrl,
sqlDatabases.get(dbUrl));
        }
    }

    private void truncateNeo4jDatabaseWork() {
        try (org.neo4j.driver.Driver driver =
createNeo4jDriver();
             Session session =
driver.session()) {

            session.run("MATCH (w:work) DETACH
DELETE w");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void truncateSqlDatabasesWork() {
        for (String dbUrl :
sqlDatabases.keySet()) {
            executeWorkTruncateBatch(dbUrl,
sqlDatabases.get(dbUrl));
        }
    }

    private void
truncateNeo4jDatabaseCustomer() {
        try (org.neo4j.driver.Driver driver =
createNeo4jDriver();
             Session session =
driver.session()) {

            session.run("MATCH (c:customer)
DETACH DELETE c");
            session.run("MATCH (i:invoice)
DETACH DELETE i");
            session.run("MATCH (t:target)
DETACH DELETE t");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void
truncateSqlDatabasesCustomer() {
        for (String dbUrl :
sqlDatabases.keySet()) {

executeCustomerTruncateBatch(dbUrl,
sqlDatabases.get(dbUrl));
        }
    }
```

```
    private void executeTruncateBatch(String
dbUrl, String[] dbInfo) {
        try (Connection conn =
createConnection(dbUrl, dbInfo);
             Statement stmt =
conn.createStatement()) {

            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=0;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.customer;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.invoice;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.work;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.workhours;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.workinvoice;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.worktarget;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.target;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.useditem;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.worktype;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.item;");
            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=1;");
            stmt.executeBatch();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void
executeWorkTruncateBatch(String dbUrl,
String[] dbInfo) {
        try (Connection conn =
createConnection(dbUrl, dbInfo);
             Statement stmt =
conn.createStatement()) {

            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=0;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.work;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.useditem;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.workhours;");
            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=1;");
            stmt.executeBatch();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void
executeCustomerTruncateBatch(String dbUrl,
String[] dbInfo) {
        try (Connection conn =
createConnection(dbUrl, dbInfo);
             Statement stmt =
conn.createStatement()) {

            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=0;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.workinvoice;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.worktarget;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.target;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.invoice;");
            stmt.addBatch("TRUNCATE TABLE
warehouse.customer;");
            stmt.addBatch("SET
FOREIGN_KEY_CHECKS=1;");
            stmt.executeBatch();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private org.neo4j.driver.Driver
createNeo4jDriver() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        return
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));
    }
}
```

2.4 DataExecutor.java

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class DataGenerator {

    // ... Other members

    public void createSampleTables(String
dbUrl) {
        String[] dbSettings =
sqlDatabases.get(dbUrl);
        String database = "testdata";
        String dropDatabase = "DROP DATABASE
`" + database + "`";
        String createDatabase = "CREATE
DATABASE IF NOT EXISTS `" + database + "`";

        String firstnames = "CREATE TABLE IF
NOT EXISTS `firstnames` ("
                + "`id` serial,"
                + "`firstname` varchar(100)
NOT NULL,"
                + "PRIMARY KEY (`id`))";

        String surnames = "CREATE TABLE IF NOT
EXISTS `surnames` ("
                + "`id` serial,"
                + "`surname` varchar(100) NOT
```

```
NULL,"
                + "PRIMARY KEY (`id`))";

        String addresses = "CREATE TABLE IF
NOT EXISTS `addresses` ("
                + "`id` serial,"
                + "`street` varchar(200) NOT
NULL,"
                + "`city` varchar(100) NOT
NULL,"
                + "`district` varchar(100) NOT
NULL,"
                + "`region` varchar(50) NOT
NULL,"
                + "`postcode` varchar(50) NOT
NULL,"
                + "PRIMARY KEY (`id`))";

        executeSQLUpdate(dropDatabase,
"jdbc:mariadb://127.0.0.1/", dbSettings);
        executeSQLUpdate(createDatabase,
"jdbc:mariadb://127.0.0.1/", dbSettings);
        executeSQLUpdate(firstnames,
"jdbc:mariadb://127.0.0.1/" + database,
dbSettings);
        executeSQLUpdate(surnames,
"jdbc:mariadb://127.0.0.1/" + database,
dbSettings);
        executeSQLUpdate(addresses,
"jdbc:mariadb://127.0.0.1/" + database,
dbSettings);
    }

    public void getSampleData() {
        try {
            firstnames = fetchData("SELECT
firstname FROM testdata.firstnames;");
            surnames = fetchData("SELECT
surname FROM testdata.surnames;");
            addresses =
fetchAddressData("SELECT street, city,
district, region, postcode FROM
testdata.addresses;");

        } catch (Exception e) {
            handleException(e);
        }
    }

    public void printSampleDataSizes() {
        Logger.Log("Firstnames size: " +
firstnames.size());
        Logger.Log("Surnames size: " +
surnames.size());
        Logger.Log("Addresses size: " +
addresses.size());
    }

    // Additional methods for fetching counts
and last IDs...

    private List<String> fetchData(String
query) throws SQLException {
        List<String> data = new ArrayList<>();
        ResultSet rs = executeSQLQuery(query);

        while (rs.next()) {
            data.add(rs.getString(1));
        }

        return data;
    }

    private List<HashMap<String, String>>
fetchAddressData(String query) throws
SQLException {
        List<HashMap<String, String>> data =
new ArrayList<>();
        ResultSet rs = executeSQLQuery(query);

        while (rs.next()) {
            HashMap<String, String> address =
new HashMap<>();
            address.put("street",
rs.getString("street"));
            address.put("city",
rs.getString("city"));
            address.put("district",
rs.getString("district"));
            address.put("region",
rs.getString("region"));
            address.put("postcode",
rs.getString("postcode"));
            data.add(address);
        }

        return data;
    }

    private void handleException(Exception e)
{
        System.err.println("Exception: " +
e.getMessage());
        e.printStackTrace();
    }
}
```

2.5 LogWrapper.java

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Logger {

    static String _filename = "";
    static File _file;
    static FileWriter _myWriter;

    public static void Log(String message) {

        try {
            String filename = _getFileName();
            FileWriter fw = new
FileWriter(filename, true); // the true will
append the new data
            fw.write(message);// appends the
string to the file
            fw.close();
        } catch (IOException ioe) {
            System.err.println("IOException: "
+ ioe.getMessage());
        }
    }

    static String _getFileName() {

        if (!_filename.isEmpty())
            return _filename;

        DateTimeFormatter dtf =
```

```
DateTimeFormatter.ofPattern("HH:mm:ss");
        LocalDateTime now =
LocalDateTime.now();

        String fileName =
"_logger_output.txt";
        String osPrefix =
"/Users/pbasha/output/";
        String timePrefix = dtf.format(now);

        _filename = osPrefix + timePrefix +
fileName;

        return _filename;
    }

    static void _createFileIfNotExist() {
        try {
            if (_file == null) {
                _file = new
File(_getFileName());
                _file.createNewFile();
            }

            _myWriter = new
FileWriter(_getFileName(), true);

        } catch (IOException e) {
            Logger.Log("An error occurred.");
            e.printStackTrace();
        }
    }

    static void Close() {
        try {
            _myWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.6 DataStamp.java

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;

import java.sql.*;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.ZoneId;
import java.util.*;
import
java.util.concurrent.locks.ReentrantLock;

public class DataGeneratorThreadCustomer
extends Thread {

    private final HashMap<String, String[]>
sqlDatabases;
    private final HashMap<String, String>
neo4jSettings;

    private int iterationCount = 0;
    private int batchExecuteValue = 0;
    private int invoiceFactor = 0;
    private int targetFactor = 0;
    private int workFactor = 0;
    private int sequentialInvoices = 0;
    private int workCount = 0;

    private int threadIndex = 0;
    private int customerIndex = 0;
    private int invoiceIndex = 0;
    private int targetIndex = 0;

    private int firstnameIndex = 0;
    private int surnameIndex = 0;
    private int addressIndex = 0;

    private final ReentrantLock lock;

    private List<String> firstnames;
    private List<String> surnames;
    private List<HashMap<String, String>>
addresses;

    public DataGeneratorThreadCustomer(int
threadIndex, int iterationCount, int
batchExecuteValue,

HashMap<String, String[]> sqlDatabases,

HashMap<String, String> neo4jSettings,

ReentrantLock lock, int invoiceFactor, int
targetFactor, int workFactor,
                                        int
sequentialInvoices, List<String> firstnames,
List<String> surnames,

List<HashMap<String, String>> addresses, int
customerIndex, int invoiceIndex,
                                        int
targetIndex, int workCount) {
        this.threadIndex = threadIndex;
        this.iterationCount = iterationCount;
        this.batchExecuteValue =
batchExecuteValue;
        this.invoiceFactor = invoiceFactor;
        this.targetFactor = targetFactor;
        this.workFactor = workFactor;
        this.sequentialInvoices =
sequentialInvoices;
        this.firstnames = firstnames;
        this.surnames = surnames;
        this.addresses = addresses;
        this.customerIndex = customerIndex;
        this.invoiceIndex = invoiceIndex;
        this.targetIndex = targetIndex;
        this.workCount = workCount;
        this.sqlDatabases = sqlDatabases;
        this.neo4jSettings = neo4jSettings;
        this.lock = lock;
    }

    public void run() {
        try {
            initializeDatabaseConnections();
            generateData();
            closeDatabaseConnections();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void
initializeDatabaseConnections() throws
SQLException, ClassNotFoundException {
        initializeSqlDatabaseConnections();
        initializeNeo4jConnection();
    }
```

```
    private void
initializeSqlDatabaseConnections() throws
SQLException, ClassNotFoundException {
        for (String dbUrl :
sqlDatabases.keySet()) {
            String[] dbInfo =
sqlDatabases.get(dbUrl);
            String dbDriver = dbInfo[0];
            String dbUsername = dbInfo[1];
            String dbPassword = dbInfo[2];

            Class.forName(dbDriver);

            Connection connection =
DriverManager.getConnection(dbUrl, dbUsername,
dbPassword);
            connectionList.add(connection);

            prepareSqlStatements(connection);
        }
    }

    private void initializeNeo4jConnection() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        driver =
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));

        session = driver.session();
    }

    private void
prepareSqlStatements(Connection connection)
throws SQLException {
        PreparedStatement customerStatement =
connection.prepareStatement("INSERT INTO
warehouse.customer (id, name, address) VALUES
(?,?,?)");
        PreparedStatement invoiceStatement =
connection.prepareStatement("INSERT INTO
warehouse.invoice (id, customerId, state,
duedate, previousinvoice) VALUES
(?,?,?,?,?)");
        PreparedStatement workInvoiceStatement
= connection.prepareStatement("INSERT INTO
warehouse.workinvoice (workId, invoiceId)
VALUES (?,?)");
        PreparedStatement targetStatement =
connection.prepareStatement("INSERT INTO
warehouse.target (id, name, address,
customerid) VALUES (?,?,?,?)");
        PreparedStatement workTargetStatement
= connection.prepareStatement("INSERT INTO
warehouse.worktarget (workId, targetId) VALUES
(?,?)");

        HashMap<String, PreparedStatement>
preparedStatements = new HashMap<>();

        preparedStatements.put("customer",
customerStatement);
        preparedStatements.put("invoice",
invoiceStatement);
        preparedStatements.put("target",
targetStatement);
        preparedStatements.put("workinvoice",
workInvoiceStatement);
        preparedStatements.put("worktarget",
workTargetStatement);


preparedStatementsList.add(preparedStatements)
;
    }

    private void generateData() throws
SQLException, InterruptedException {
        for (int iterator = 0; iterator <
iterationCount; iterator++) {
            insertCustomer(iterator,
batchExecuteValue);
        }
    }

    private void insertCustomer(int iterator,
int batchExecuteValue) throws SQLException,
InterruptedException {
        prepareCustomerData();
        insertCustomerData();
        if (shouldExecuteBatch(iterator)) {
            executeBatch();
        }
    }

    private void prepareCustomerData() {
        setIndexes(customerIndex);

        String name =
firstnames.get(firstnameIndex) + " " +
surnames.get(surnameIndex);

        String streetAddress =
addresses.get(addressIndex).get("street") + "
" +

addresses.get(addressIndex).get("city") + " "
+

addresses.get(addressIndex).get("district") +
" " +

addresses.get(addressIndex).get("region") + "
" +

addresses.get(addressIndex).get("postcode");

        sqlInsert = "INSERT INTO
warehouse.customer (id, name, address) VALUES
(" +
                customerIndex + ",\"" + name +
"\",\"" + streetAddress + "\")";
    }

    private void insertCustomerData() throws
SQLException, InterruptedException {
        PreparedStatement customerStatement =
getPreparedStatement("customer");

        customerStatement.setInt(1,
customerIndex);
        customerStatement.setString(2, name);
        customerStatement.setString(3,
streetAddress);
        customerStatement.addBatch();

        writeToNeo4J(session,
buildNeo4JQuery());
        prepareInvoiceData();
    }

    private boolean shouldExecuteBatch(int
iterator) {
        return iterator % batchExecuteValue ==
0 || iterator == (iterationCount - 1);
    }

    private void executeBatch() throws
SQLException {
        for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
```

```
            for (PreparedStatement statement :
preparedStatements.values()) {
                statement.executeBatch();
            }
        }
    }

    private PreparedStatement
getPreparedStatement(String key) {
        return preparedStatementsList.stream()
                .map(map -> map.get(key))
                .findFirst()
                .orElseThrow(() -> new
IllegalArgumentException("Invalid key: " +
key));
    }
}
```

2.7 DataWarehouse.java

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;

import java.sql.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Random;
import
java.util.concurrent.locks.ReentrantLock;

public class
DataGeneratorThreadItemsAndWorkTypes extends
Thread {

    private HashMap<String, String[]>
sqlDatabases;
    private HashMap<String, String>
neo4jSettings;

    private int batchExecuteValue = 0;
    private int threadIndex = 0;
    private final int INITIALITEMINDEX;
    private final int INITIALWORKTYPEINDEX;

    private int itemIndex = 0;
    private int workTypeIndex = 0;
    private int itemCount = 0;
    private int workTypeCount = 0;

    private ReentrantLock lock;

    public
DataGeneratorThreadItemsAndWorkTypes(int
threadIndex, int batchExecuteValue,
HashMap<String, String[]> sqlDatabases,

HashMap<String, String> neo4jSettings,
ReentrantLock lock,

int itemIndex, int itemCount, int
workTypeIndex, int workTypeCount) {

        this.threadIndex = threadIndex;
        this.batchExecuteValue =
batchExecuteValue;
        this.itemIndex = itemIndex;
        this.INITIALITEMINDEX = itemIndex;
        this.itemCount = itemCount;
        this.workTypeIndex = workTypeIndex;
        this.INITIALWORKTYPEINDEX =
workTypeIndex;
        this.workTypeCount = workTypeCount;
        this.sqlDatabases = sqlDatabases;
        this.neo4jSettings = neo4jSettings;
        this.lock = lock;
    }

    public void run() {
        try {
            initializeConnections();
            generateItemsAndWorkTypes();
            closeConnections();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void initializeConnections()
throws SQLException, ClassNotFoundException {
        initializeSqlDatabaseConnections();
        initializeNeo4jConnection();
    }

    private void
initializeSqlDatabaseConnections() throws
SQLException, ClassNotFoundException {
        for (String dbUrl :
sqlDatabases.keySet()) {
            String[] dbInfo =
sqlDatabases.get(dbUrl);
            String dbDriver = dbInfo[0];
            String dbUsername = dbInfo[1];
            String dbPassword = dbInfo[2];

            Class.forName(dbDriver);

            Connection connection =
DriverManager.getConnection(dbUrl, dbUsername,
dbPassword);
            connectionList.add(connection);

            prepareSqlStatements(connection);
        }
    }

    private void initializeNeo4jConnection() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        org.neo4j.driver.Driver driver =
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));
        session = driver.session();
    }

    private void
prepareSqlStatements(Connection connection)
throws SQLException {
        PreparedStatement itemStatement =
connection.prepareStatement(
                "INSERT INTO warehouse.item
(id, name, balance, unit, purchaseprice, vat,
removed) VALUES (?,?,?,?,?,?,?)");
```

```
        PreparedStatement workTypeStatement =
connection.prepareStatement(
                "INSERT INTO
warehouse.worktype (id, name, price) VALUES
(?, ?, ?)");

        HashMap<String, PreparedStatement>
preparedStatements = new HashMap<>();

        preparedStatements.put("item",
itemStatement);
        preparedStatements.put("worktype",
workTypeStatement);


preparedStatementsList.add(preparedStatements)
;
    }

    private void generateItemsAndWorkTypes()
throws SQLException, InterruptedException {
        generateItems();
        generateWorkTypes();
    }

    private void generateItems() throws
SQLException, InterruptedException {
        for (int iterator = 0; iterator <
itemCount; iterator++) {
            insertItems(iterator,
batchExecuteValue, session,
preparedStatementsList);
            itemIndex++;
        }
    }

    private void generateWorkTypes() throws
SQLException, InterruptedException {
        for (int iterator = 0; iterator <
workTypeCount; iterator++) {
            insertWorkTypes(iterator,
batchExecuteValue, session,
preparedStatementsList);
            workTypeIndex++;
        }
    }

    private void closeConnections() throws
SQLException {
        for (Connection connection :
connectionList) {
            connection.close();
        }
        session.close();
        driver.close();
    }
}
```

2.8 Models.java

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;

import java.sql.*;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.ZoneId;
import java.util.*;
import
java.util.concurrent.locks.ReentrantLock;

public class
DataGeneratorThreadSequentialInvoices extends
Thread {

    private HashMap<String, String[]>
sqlDatabases;
    private HashMap<String, String>
neo4jSettings;

    private int batchExecuteValue = 0;
    private int firstInvoiceIndex = 0;
    private int sequentialInvoiceCount = 0;
    private int threadIndex = 0;
    private int customerIndex = 0;
    private int invoiceIndex = 0;
    private ReentrantLock lock;

    public
DataGeneratorThreadSequentialInvoices(int
threadIndex, int batchExecuteValue,

HashMap<String, String[]> sqlDatabases,

HashMap<String, String> neo4jSettings,
ReentrantLock lock,

int sequentialInvoiceCount, int customerIndex,
int invoiceIndex, int firstInvoiceIndex) {
        this.threadIndex = threadIndex;
        this.batchExecuteValue =
batchExecuteValue;
        this.sequentialInvoiceCount =
sequentialInvoiceCount;
        this.customerIndex = customerIndex;
        this.invoiceIndex = invoiceIndex;
        this.firstInvoiceIndex =
firstInvoiceIndex;
        this.sqlDatabases = sqlDatabases;
        this.neo4jSettings = neo4jSettings;
        this.lock = lock;
    }

    public void run() {
        try {
            initializeConnections();
            generateSequentialInvoices();
            closeConnections();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void initializeConnections()
throws SQLException, ClassNotFoundException {
        initializeSqlDatabaseConnections();
        initializeNeo4jConnection();
    }

    private void
initializeSqlDatabaseConnections() throws
SQLException, ClassNotFoundException {
        for (String dbUrl :
sqlDatabases.keySet()) {
            String[] dbInfo =
sqlDatabases.get(dbUrl);
            String dbDriver = dbInfo[0];
            String dbUsername = dbInfo[1];
            String dbPassword = dbInfo[2];
```

```
            Class.forName(dbDriver);

            Connection connection =
DriverManager.getConnection(dbUrl, dbUsername,
dbPassword);
            connectionList.add(connection);

            prepareSqlStatements(connection);
        }
    }

    private void initializeNeo4jConnection() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        org.neo4j.driver.Driver driver =
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));
        session = driver.session();
    }

    private void
prepareSqlStatements(Connection connection)
throws SQLException {
        PreparedStatement invoiceStatement =
connection.prepareStatement(
                "INSERT INTO warehouse.invoice
(id, customerId, state, duedate,
previousinvoice) VALUES (?,?,?,?,?)");

        HashMap<String, PreparedStatement>
preparedStatements = new HashMap<>();
        preparedStatements.put("invoice",
invoiceStatement);

preparedStatementsList.add(preparedStatements)
;
    }

    private void generateSequentialInvoices()
throws SQLException, InterruptedException {
        for (int iterator = 0; iterator <
sequentialInvoiceCount; iterator++) {
            insertSequentialInvoices(iterator,
batchExecuteValue, session,
preparedStatementsList);
            invoiceIndex++;
        }
    }

    private void closeConnections() throws
SQLException {
        for (Connection connection :
connectionList) {
            connection.close();
        }
        session.close();
        driver.close();
    }

    // Other existing methods...

    private void insertSequentialInvoices(int
iterator, int batchExecuteValue, Session
session,

List<HashMap> preparedStatementsList) throws
SQLException, InterruptedException {
        PreparedStatement invoice;

        String sqlInsert;
        String cypherCreate;

        Random random = new Random();

        int state = 1 + random.nextInt(3);

        GregorianCalendar gregorianCalendar =
new GregorianCalendar();
        int year =
Calendar.getInstance().get(Calendar.YEAR);

gregorianCalendar.set(gregorianCalendar.YEAR,
year);

        int dayOfYear = 1 +
random.nextInt(gregorianCalendar.getActualMaxi
mum(gregorianCalendar.DAY_OF_YEAR));

gregorianCalendar.set(gregorianCalendar.DAY_OF
_YEAR, dayOfYear);

        Date dueDate =
gregorianCalendar.getTime();
        java.sql.Date sqlDueDate = new
java.sql.Date(dueDate.getTime());
        DateFormat dateFormat = new
SimpleDateFormat("dd-MM-yyyy");
        String dueDateAsString =
dateFormat.format(dueDate);

        for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
            invoice =
preparedStatements.get("invoice");

            invoice.setInt(1, invoiceIndex);
            invoice.setInt(2, customerIndex);
            invoice.setInt(3, state);
            invoice.setDate(4, sqlDueDate,
gregorianCalendar);

            if (invoiceIndex ==
firstInvoiceIndex) {
                sqlInsert = "INSERT INTO
warehouse.invoice (id, customerId, state,
duedate, previousinvoice) VALUES ("
                        + invoiceIndex + "," +
customerIndex + "," + state + ",STR_TO_DATE('"
+ dueDateAsString
                        + "','%d-%m-%Y')," +
invoiceIndex + ")";
                invoice.setInt(5,
invoiceIndex);
            } else {
                sqlInsert = "INSERT INTO
warehouse.invoice (id, customerId, state,
duedate, previousinvoice) VALUES ("
                        + invoiceIndex + "," +
customerIndex + "," + state + ",STR_TO_DATE('"
+ dueDateAsString
                        + "','%d-%m-%Y')," +
(invoiceIndex - 1) + ")";
                invoice.setInt(5, invoiceIndex
- 1);
            }

            invoice.addBatch();
        }

        LocalDate localDate =
dueDate.toInstant().atZone(ZoneId.systemDefaul
t()).toLocalDate();
        int month = localDate.getMonthValue();
        int day = localDate.getDayOfMonth();

        if (invoiceIndex == firstInvoiceIndex)
{
            cypherCreate = "CREATE (1:invoice
{invoiceId: " + invoiceIndex + ", customerId:
" + customerIndex
```

```
                    + ", state: " + state + ",
duedate: \"date({ year:" + year + ", month:" +
month + ", day:" + day
                    + " })\", previousinvoice:
" + invoiceIndex + "})";
            writeToNeo4J(session,
cypherCreate);

            cypherCreate = "MATCH
(a:customer),(l:invoice) WHERE a.customerId =
" + customerIndex
                    + " AND l.invoiceId = " +
invoiceIndex + " CREATE (a)-[m:PAYS]->(l)";
            writeToNeo4J(session,
cypherCreate);
        } else {
            cypherCreate = "CREATE (l:invoice
{invoiceId: " + invoiceIndex + ", customerId:
" + customerIndex
                    + ", state: " + state + ",
duedate: \"date({ year:" + year + ", month:" +
month + ", day:" + day
                    + " })\", previousinvoice:
" + (invoiceIndex - 1) + "})";
            writeToNeo4J(session,
cypherCreate);

            cypherCreate = "MATCH
(a:customer),(l:invoice) WHERE a.customerId =
" + customerIndex
                    + " AND l.invoiceId = " +
invoiceIndex + " CREATE (a)-[m:PAYS]->(l)";
            writeToNeo4J(session,
cypherCreate);

            cypherCreate = "MATCH
(a:invoice),(b:invoice) WHERE a.invoiceId = "
+ (invoiceIndex - 1) + " AND b.invoiceId = "
                    + invoiceIndex + " CREATE
(a)-[m:PREVIOUS_INVOICE]->(b)";
            writeToNeo4J(session,
cypherCreate);
        }

        int discountPercent = 1 +
random.nextInt(101);
        double discount = (0.01 *
discountPercent);

        if (iterator % batchExecuteValue == 0
|| iterator == (sequentialInvoiceCount - 1)) {
            for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
                invoice =
preparedStatements.get("invoice");
                invoice.executeBatch();
            }
        }
    }
}
```

2.4 DataStratagerFlow.java

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Session;

import java.sql.*;
import java.util.*;
import
java.util.concurrent.locks.ReentrantLock;

public class DataGeneratorThreadWork extends
Thread {

    private final HashMap<String, String[]>
sqlDatabases;
    private final HashMap<String, String>
neo4jSettings;

    private int iterationCount = 0;
    private int batchExecuteValue = 0;

    private int threadIndex = 0;

    private int workIndex = 0;
    private int itemFactor = 0;
    private int itemCount = 0;
    private int workTypeFactor = 0;
    private int workTypeCount = 0;

    private final ReentrantLock lock = new
ReentrantLock();

    public DataGeneratorThreadWork(int
threadIndex, int iterationCount, int
batchExecuteValue,

HashMap<String, String[]> sqlDatabases,

HashMap<String, String> neo4jSettings,
                                   int
workIndex, int itemFactor, int itemCount,
                                   int
workTypeFactor, int workTypeCount) {
        this.threadIndex = threadIndex;
        this.iterationCount = iterationCount;
        this.batchExecuteValue =
batchExecuteValue;
        this.workIndex = workIndex;
        this.itemFactor = itemFactor;
        this.itemCount = itemCount;
        this.workTypeFactor = workTypeFactor;
        this.workTypeCount = workTypeCount;
        this.sqlDatabases = sqlDatabases;
        this.neo4jSettings = neo4jSettings;
    }

    public void run() {
        try {
            initializeConnections();
            generateWorkData();
            closeConnections();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void initializeConnections()
throws SQLException, ClassNotFoundException {
        initializeSqlDatabaseConnections();
        initializeNeo4jConnection();
    }

    private void
initializeSqlDatabaseConnections() throws
SQLException, ClassNotFoundException {
        for (String dbUrl :
sqlDatabases.keySet()) {
            String[] dbInfo =
sqlDatabases.get(dbUrl);
```

```
            String dbDriver = dbInfo[0];
            String dbUsername = dbInfo[1];
            String dbPassword = dbInfo[2];

            Class.forName(dbDriver);

            Connection connection =
DriverManager.getConnection(dbUrl, dbUsername,
dbPassword);
            connectionList.add(connection);

            prepareSqlStatements(connection);
        }
    }

    private void initializeNeo4jConnection() {
        String neo4jDbUrl =
neo4jSettings.get("NEO4J_DB_URL");
        String neo4jUsername =
neo4jSettings.get("NEO4J_USERNAME");
        String neo4jPassword =
neo4jSettings.get("NEO4J_PASSWORD");

        org.neo4j.driver.Driver driver =
GraphDatabase.driver(neo4jDbUrl,
AuthTokens.basic(neo4jUsername,
neo4jPassword));
        session = driver.session();
    }

    private void
prepareSqlStatements(Connection connection)
throws SQLException {
        PreparedStatement workStatement =
connection.prepareStatement("INSERT INTO
warehouse.work (id, name) VALUES (?, ?)");
        PreparedStatement usedItemStatement =
connection.prepareStatement("INSERT INTO
warehouse.useditem (amount, discount, workId,
itemId) VALUES(?, ?, ?, ?)");
        PreparedStatement workHoursStatement =
connection.prepareStatement("INSERT INTO
warehouse.workhours (worktypeId, hours,
discount, workId) VALUES(?, ?, ?, ?)");

        HashMap<String, PreparedStatement>
preparedStatements = new HashMap<>();
        preparedStatements.put("work",
workStatement);
        preparedStatements.put("useditem",
usedItemStatement);
        preparedStatements.put("workhours",
workHoursStatement);


preparedStatementsList.add(preparedStatements)
;
    }

    private void generateWorkData() throws
SQLException, InterruptedException {
        for (int iterator = 0; iterator <
iterationCount; iterator++) {
            insertWork(iterator,
batchExecuteValue, session,
preparedStatementsList);
        }
    }

    private void closeConnections() throws
SQLException {
        for (Connection connection :
connectionList) {
            connection.close();
        }
        session.close();
        driver.close();
    }

    private void writeToNeo4J(Session session,
String cypherQuery) throws SQLException {
        session.writeTransaction(tx ->
tx.run(cypherQuery));
    }

    private List<Integer> getItemIndexes(int
index) {
        List<Integer> allItemIndexes = new
ArrayList<>();
        for (int i = 0; i < itemCount; i++) {
            allItemIndexes.add(i);
        }

        Collections.shuffle(allItemIndexes,
new Random(index));

        return allItemIndexes.subList(0,
itemFactor);
    }

    private List<Integer>
getWorkTypeIndexes(int index) {
        List<Integer> allWorkTypeIndexes = new
ArrayList<>();
        for (int i = 0; i < workTypeCount;
i++) {
            allWorkTypeIndexes.add(i);
        }


Collections.shuffle(allWorkTypeIndexes, new
Random(index));

        return allWorkTypeIndexes.subList(0,
workTypeFactor);
    }

    private void insertWork(int iterator, int
batchExecuteValue, Session session,
List<HashMap> preparedStatementsList) throws
SQLException, InterruptedException {
        PreparedStatement work;
        PreparedStatement usedItem;
        PreparedStatement workHours;

        Logger.Log("Thread: " + threadIndex +
" workIndex: " + workIndex);

        int workIndexOriginal = workIndex;
        String workName = "Generic " +
workIndex;

        for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
            work =
preparedStatements.get("work");
            work.setInt(1, workIndex);
            work.setString(2, workName);
            work.addBatch();
        }

        String cypherCreate = "CREATE (s:work
{workId: " + workIndex + ", name: \"" +
workName + "\"})";
        writeToNeo4J(session, cypherCreate);

        Random r = new Random(workIndex);
        int discountPercent = 1 +
r.nextInt(101);
        double discount = 0.01 *
discountPercent;

        List<Integer> itemIndexes =
getItemIndexes(workIndex);
        insertUsedItems(itemIndexes, session,
preparedStatementsList);
```

```
        List<Integer> workTypeIndexes =
getWorkTypeIndexes(workIndex);
        insertWorkHours(workTypeIndexes,
session, preparedStatementsList);

        workIndex++;

        if (iterator % batchExecuteValue == 0
|| iterator == (iterationCount - 1)) {

executeBatchStatements(preparedStatementsList)
;
        }
    }

    private void insertUsedItems(List<Integer>
itemIndexes, Session session, List<HashMap>
preparedStatementsList) throws SQLException,
InterruptedException {
        PreparedStatement usedItem;
        for (int i = 0; i <
itemIndexes.size(); i++) {
            int itemId = itemIndexes.get(i);
            Random r = new Random(workIndex);
            int amount = 1 + r.nextInt(101);

            for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
                usedItem =
preparedStatements.get("useditem");
                usedItem.setInt(1, amount);
                usedItem.setDouble(2,
discount);
                usedItem.setInt(3, workIndex);
                usedItem.setInt(4, itemId);
                usedItem.addBatch();
            }

            String cypherCreate = "MATCH
(s:work),(v:item) WHERE s.workId=" + workIndex
+ " AND v.itemId=" + itemId + " CREATE (s)-
[ui:USED_ITEM {amount:" + amount + ",
discount:" + discount + "}]->(v) ";
            writeToNeo4J(session,
cypherCreate);

            cypherCreate = "MATCH
(v:item),(s:work) WHERE v.itemId=" + itemId +
" AND s.workId=" + workIndex + "   CREATE (v)-
[ui:USED_ITEM {amount:" + amount + ",
discount:" + discount + "}]->(s)";
            writeToNeo4J(session,
cypherCreate);
        }
    }

    private void insertWorkHours(List<Integer>
workTypeIndexes, Session session,
List<HashMap> preparedStatementsList) throws
SQLException, InterruptedException {
        PreparedStatement workHours;
        for (int i = 0; i <
workTypeIndexes.size(); i++) {
            int worktypeId =
workTypeIndexes.get(i);
            Random r = new Random(workIndex);
            int hours = r.nextInt(100);

            for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
                workHours =
preparedStatements.get("workhours");
                workHours.setInt(1,
worktypeId);
                workHours.setInt(2, hours);
                workHours.setDouble(3,
discount);
                workHours.setInt(4,
workIndex);
                workHours.addBatch();
            }

            String cypherCreate = "MATCH
(w:work),(wt:worktype) WHERE w.workId=" +
workIndex + " AND wt.worktypeId=" + worktypeId
+ " CREATE (w)-[wh:WORKHOURS {hours:" + hours
+ ", discount:" + discount + "}]->(wt) ";
            writeToNeo4J(session,
cypherCreate);

            cypherCreate = "MATCH
(wt:worktype),(w:work) WHERE wt.worktypeId=" +
worktypeId + " AND w.workId=" + workIndex  + "
CREATE (wt)-[wh:WORKHOURS {hours:" + hours +
", discount:" + discount + "}]->(w)";
            writeToNeo4J(session,
cypherCreate);
        }
    }

    private void
executeBatchStatements(List<HashMap>
preparedStatementsList) throws SQLException {
        for (HashMap<String,
PreparedStatement> preparedStatements :
preparedStatementsList) {
            PreparedStatement work =
preparedStatements.get("work");
            PreparedStatement usedItem =
preparedStatements.get("useditem");
            PreparedStatement workHours =
preparedStatements.get("workhours");

            work.executeBatch();
            usedItem.executeBatch();
            workHours.executeBatch();
        }
    }
}
```