Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Батальського Микити Аркадійовича*
(ПІБ)

академічної групи *121М-22-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

освітньої програми *Інженерія програмного забезпечення*
(назва освітньої програми)

на тему: *Розробка та дослідження ефективності впровадження технологій та методів взаємодії з базами даних*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*к.т.н. Приходченко С. Д.*	80	добре	
спеціального розділу	*к.т.н. Приходченко С. Д.*	80	добре	

Рецензент	*Доц. Малієнко А.В.*	80	добре	

Нормоконтролер	*Проф. Лактіонов І.С.*	75	добре	

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » _____ 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121м-22-1* *Батальському Микиті Аркадійовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка та дослідження ефективності впровадження технологій та методів взаємодії з базами даних*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – ефективність та якість взаємодії між Java-додатками і базами даних.

Предмет досліджень – технології та інструменти, які використовуються для взаємодії з базами даних в Java (такі як JDBC, Hibernate).

Мета НДР – виявлення переваг та недоліків методів підключення та взаємодії Java-застосунку з БД.

Вихідні дані для проведення роботи – приклад коду, який використовує JDBC та Hibernate для взаємодії з базами даних. Він послужить прикладом для аналізу та порівняння технологій.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень полягає в обґрунтуванні та розв'язанні проблеми вибору метода підключення та взаємодії між Java-застосунком та базою даних, що дасть можливість більш точно, ефективно і економно (у плані ресурсів) проектувати.

Практична цінність полягає в виявленні найбільш оптимальної технології для взаємодії java-застосунку з БД.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подані у вигляді, що дозволяє побачити та оцінити безпосереднє використання запропонованих методів. В результаті роботи повинне бути розроблене програмне забезпечення, що буде наглядно демонструвати переваги та недоліки способів взаємодії з базами даних на мові програмування Java.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2023-30.09.2023
Розробка моделі, метода та програмного забезпечення розрахунку температурних полів в двовимірних тілах	01.10.2023-15.11.2023
Використання програми та аналіз отриманих результатів	16.11.2023-12.12.2023

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки можливості для нових проектів правильно обрати потрібну їм технологію, згідно з очікуваних результатів, що забезпечить їм економію часу та трудового ресурсу, якби потрібно було змінити підхід вже розробленого застосунку.

Соціальний ефект від реалізації результатів роботи очікується позитивним, завдяки полегшенню вибору потрібної технології при розробках Java-застосунків.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Приходченко С.Д.*.
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Батальський М.А.*.
(підпис) (прізвище, ініціали)

Дата видачі завдання: *25.10.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *17.12.2023*

РЕФЕРАТ

Пояснювальна записка: 93 с., 54 рис., 4 табл., 2 дод., 35 джерел.

Об'єкт досліджень: ефективність та якість взаємодії між Java-додатками і базами даних.

Предмет досліджень: технології та інструменти, які використовуються для взаємодії з базами даних в Java (такі як JDBC, Hibernate).

Мета кваліфікаційної роботи: виявлення переваг та недоліків методів підключення та взаємодії Java-застосунку з БД.

Методи дослідження: реалізація технологій jdbc та hibernate та порівняння їх за допомогою написаних методів застосунку, заміру часу виконання, використовуючи БД PostgreSQL, порівняти витрачені ресурси під час виконання програми, а також порівняти витрату ресурсів java virtual machine за допомогою застосунку jconsole.

Наукова новизна: обґрунтування та розв’язання проблеми вибору метода підключення та взаємодії між Java-застосунком та базою даних.

Практична цінність: вибір найбільш оптимальної технології для взаємодії java-застосунку з БД.

Область застосування: визначення найбільш оптимальної технології для конкретного проекту.

Економічний ефект: можливість для нових проектів правильно обрати потрібну їм технологію, згідно з очікуваних результатів, що забезпечить їм економію часу та ресурсів.

Прогнозні припущення про розвиток досліджень: перспективним напрямком розвитку даної роботи може стати доповнення у вигляді транзакцій, кешування.

Ключові слова: JAVA, БАЗИ ДАНИХ, JDBC, HIBERNATE, АНАЛІЗ ТЕХНОЛОГІЙ, ВЗАЄМОДІЯ, ПРОГРАМА.

ABSTRACT

Explanatory note: 93 p., 54 figs., 4 tabl., 2 appx., 35 sources.

Object of research: efficiency and quality of interaction between Java applications and databases.

Research subject: technologies and tools used to interact with databases in Java (such as JDBC, Hibernate).

The purpose of the qualification work: to identify the advantages and disadvantages of methods of connection and interaction of a Java application with a database.

Research methods: implementation of jdbc and hibernate technologies and their comparison using written application methods, execution time measurement using the PostgreSQL database, comparison of consumed resources during program execution, and comparison of java virtual machine resource consumption using the jconsole application.

Scientific novelty: substantiation and solution of the problem of choosing a method of connection and interaction between a Java application and a database.

Practical value: choosing the most optimal technology for the interaction of a java application with a database.

Scope: determination of the most optimal technology for a specific project.

Economic effect: the opportunity for new projects to correctly choose the technology they need, according to the expected results, which will save them time and resources.

Predictive assumptions about the development of research: a promising direction for the development of this work may be an addition in the form of transactions, caching.

Keywords: JAVA, DATABASES, JDBC, HIBERNATE, TECHNOLOGY ANALYSIS, INTERACTION, APPLICATION.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

PG – Postgres;

JDBC – Java Database Connectivity;

БД – база даних;

ООП – об'єктно-орієнтоване програмування;

ОС – операційна система;

ПК – персональний комп’ютер;

ПЗ – програмне забезпечення;

ІМ – інтернет-магазин.

ВСТУП

Сучасний розвиток інформаційних технологій та поширення баз даних в сфері різних доменів вимагає вдосконалення та оптимізації методів взаємодії з ними. Одним із ключових аспектів цього процесу є розробка ефективних інструментів для роботи з базами даних, а також вміння адаптувати їх до змінюючихся вимог та технологічних стандартів.

Однією з потужних та широко використовуваних технологій в розробці програмного забезпечення є Java. Ця мова програмування володіє великою популярністю завдяки своїй універсальності, надійності та кросплатформенності. Однак, незважаючи на ці переваги, важливо ефективно використовувати інструменти для взаємодії з базами даних у Java-застосунках.

Отже, обрана тема актуальна в контексті сучасних вимог до розробки програмного забезпечення та буде спрямована на вирішення конкретних завдань з оптимізації взаємодії Java-застосунків з базами даних з використанням технологій JDBC та Hibernate.

Мета НДР – виявлення переваг та недоліків методів підключення та взаємодії Java-застосунку з БД.

Об’єкт досліджень – ефективність та якість взаємодії між Java-додатками і базами даних.

Предмет досліджень – технології та інструменти, які використовуються для взаємодії з базами даних в Java (такі як JDBC, Hibernate).

Методи дослідження: розробка Java-застосунку, який буде використовувати технології JDBC та Hibernate для аналізу та оптимізації взаємодії з базами даних за допомогою побудованої структури та за допомогою написаних методів для тестування: масовий запис сутностей у БД, масове зчитування випадкових сутностей з БД, масове оновлення даних сутностей та масові операції по об’єднанню різних таблиць за допомогою звонішніх ключів, налаштованих у сутностях. Вивчення та порівняння цих двох методів дозволить

визначити їх переваги та недоліки в різних сценаріях використання. Такий аналіз стане важливим кроком у покращенні ефективності та продуктивності програм.

Новизна запропонованих рішень полягає в обґрунтуванні та розв’язанні проблеми вибору метода підключення та взаємодії між Java-застосунком та базою даних, що дасть можливість більш точно, ефективно і економно (у плані ресурсів) проектувати.

Практичне значення полягає в виявленні найбільш оптимальної технології для взаємодії java-застосунку з БД.

Особистий внесок автора полягає у допоміжному застосуванні та аналізу статистики, що надається програмними застосунками pgAdmin4 та jconsole [13]. Перший застосунок надає статистику у вигляді графіків щодо використаних ресурсів БД, як транзакції, сесії і т.д.. Другий надає інформацію щодо витрачених ресурсів java, як потоки, створені класи, використання пам’яті кучі і т.д..

Структура та обсяг кваліфікаційної роботи. Робота містить вступ, три розділи та висновки. Також містить дев’яносто дві сторінки друкованого тексту, п’ятдесят чотири рисунки, чотири таблиці, перелік використаних джерел та два додатки.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1. Загальні відомості з предметної галузі

Започаткована у світі інформаційних технологій революція неминуче створює необхідність у вдосконаленні та оптимізації методів взаємодії програмного забезпечення із базами даних. В цьому контексті ключове значення набуває вибір технологій, що лежать в основі цієї взаємодії. Доцільність обрання Java як основної мови програмування для розробки додатків визнається не лише за її відмінною кросплатформенністю, але й за потужністю та гнучкістю вирішення широкого спектру завдань.

Інтеграція з базами даних стає важливою складовою процесу розробки програм, особливо в умовах постійного розширення обсягів даних та зростання їх складності. Для ефективної взаємодії з базами даних у Java-застосунках стає актуальним вибір оптимальних інструментів, серед яких найбільше визнані JDBC та Hibernate.

JDBC (Java Database Connectivity) визначає стандарти для з'єднання Java-застосунків з різними типами баз даних. Його використання дозволяє розробникам ефективно взаємодіяти із реляційними базами даних, використовуючи стандартні SQL-запити. З іншого боку, Hibernate, як об'єктно-реляційна система управління базами даних, пропонує вищий рівень абстракції, спрощуючи процес роботи з даними та дозволяючи розробникам уникнути прямих SQL-запитів.

В умовах швидкого розвитку технологій та зростання обчислювальної потужності, удосконалення взаємодії з базами даних через використання Java, JDBC та Hibernate стає необхідністю. Власне розроблений застосунок покликаний стати важливим кроком у напрямку оптимізації цього взаємодії та підвищення ефективності розробки програмного забезпечення у цьому контексті.

Java є однією з найпопулярніших мов програмування в світі, і вона широко використовується для створення різноманітних програм, які потребують доступу до даних, зокрема до баз даних.

Існує кілька факторів, які роблять Java зручною та популярною мовою для роботи з базами даних:

- Java відома своєю високою безпекою та надійністю. Вона дозволяє побудувати додатки з високим рівнем захисту даних та виконання операцій над ними.
- Java є платформонезалежною мовою, що означає, що програми, написані на Java, можуть працювати на різних операційних системах без змін.
- Існує велика та активна спільнота розробників Java, яка надає доступ до багатьох корисних бібліотек та інструментів для роботи з базами даних.
- Java надає широкі можливості для роботи з різними типами баз даних, включаючи реляційні, NoSQL, і багато інших.

У якості більш підведеного прикладу програми до реальних умов у нашому світі була прийнята модель інтернет-магазину. ІМ – це вже «еволюція» продажу товарів у наш час, але спочатку все було складніше та ситуативно. Ще з давніх давен люди продавали та обмінювали товари, але тоді було багато неприємностей, наприклад: коли тобі потрібен був той, чи інший товар – ти не знав, чи є він десь, потрібно було йти та шукати його, уповаючи на вдачу, до того ж, ти не знав, у яку ціну він може бути тобі запропонований, тільки якщо ти раніше його не купляв. Це викликало багато труднощів у ті часи, проте час йде, технології розвиваються, люди вигадують нові способи продажів та інформування споживачів щодо наявності свого товару та ціни. Одним із найпоширеніших способів і є ІМ, а саме завдяки можливості використання БД у якості «складу», до того ж, з дуже простими способами дізнатись точну кількість того, чи іншого товару. Зокрема, є ще переваги даного підходу: користувач завжди може побачити актуальний статус товару (є в наявності, або ж закінчився), передивитись ціну на товар, що може бути легко змінена з боку магазину у разі потреби, і в цьому випадку користувач зможе побачити на сайті

(зазвичай сайт) вже нову ціну і вона не стане для нього «сюрпризом», коли прийде за товаром, або замовить його з доставкою. Не тільки користувач отримує переваги і легкість завдяки ІМ-ам, сам магазин витрачає менше часу на те, щоб дати користувачам інформацію щодо залишків товару, зміни ціни та решти, вся ця інформація зберігається у БД і при правильному налаштуванні будуть тільки переваги. Магазин також отримує інформацію про клієнта, яка йому потрібна і яка вказується при реєстрації, магазин також завдяки застосунку може легко отримати список замовлень того, чи іншого користувача.

1.2. Призначення розробки

Метою даної розробки є створення демонстраційного додатку на Java для порівняльного аналізу ефективності та зручності використання різних технологій доступу до баз даних: JDBC та Hibernate.

Даний продукт має мету порівняти швидкість обробки запитів до БД та використання ресурсів цими технологіями. Для цього мусить бути присутнім певний функціонал для запитів до БД.

Призначення додатку:

- Надати практичне порівняння швидкодії CRUD операцій через низькорівневий JDBC та популярний ORM framework Hibernate.
- Продемонструвати особливості роботи з даними через кожен інтерфейс програмування.
- Дати уявлення про складність реалізації типових задач доступу до даних за допомогою різних API.
- Сформувати рекомендації щодо оптимального вибору технології для конкретних прикладних задач та проектів.
- Виступити навчальним посібником для вивчення можливостей Java у сфері роботи з базами даних.

Призначення розробки полягає у наданні наочного прикладу переваг та недоліків технологій для використання у своєму проекті розробникам, що вагаються у виборі, або планують перейти на іншу технологію.

1.3. Постановка завдання

У сучасному інформаційному середовищі важливою складовою розробки програмного забезпечення є ефективна взаємодія з базами даних. Обрана тема дослідження – " Розробка та дослідження ефективності впровадження технологій та методів взаємодії з базами даних" – ставить за мету вивчення та порівняння двох ключових технологій взаємодії з базами даних в контексті розробки на мові програмування Java.

Метою даного проекту є розробка Java-застосунку, який буде використовувати технології JDBC та Hibernate для аналізу взаємодії з базами даних. Проект спрямований на створення інструменту, який дозволить розробникам ефективно взаємодіяти з різними типами баз даних, а також визначити переваги та недоліки використання JDBC та Hibernate в різних сценаріях.

Завданням є:

- розробка ПЗ для аналізу технологій взаємодії з базами даних;
- Використання мови програмування Java як основної платформи для реалізації застосунку;
- Реалізація модулів, які використовуватимуть технологію JDBC для взаємодії з реляційними базами даних;
- Розробка функціоналу, що використовуватиме Hibernate для роботи з об'єктно-реляційними властивостями даних;
- Проведення докладного аналізу ефективності та продуктивності взаємодії з базами даних через JDBC;
- Порівняння результатів з використанням Hibernate.

Кінцевий продукт представляє собою виключно BE-частину ПЗ та передбачає наступний функціонал стосовно БД:

- можливість створення сутності користувача, замовлення, продукту та частини замовлення;
- можливість редагування сутності користувача, замовлення, продукту та частини замовлення;
- можливість видалення сутності користувача, замовлення, продукту та частини замовлення;
- можливість зчитування сутності користувача, замовлення, продукту та частини замовлення.

Для розробки було обрано мову програмування JAVA, стек технологій: JDBC та Hibernate (та інші, що являються допоміжними).

1.4. Вимоги до програми або програмного виробу

Кінцевий продукт має дотримуватися наступних функціональними вимог для порівняння технологій:

- стандартні CRUD операції для всіх сутностей (користувач, замовлення, продукт, частина замовлення);
- для більш об’єктивного порівняння мають бути створені дві БД з ідентичними сутностями;
- має дотримуватись ідентичність допоміжного стеку технологій;
- додати методи для порівняння більше-менш приближені до реальних сценаріїв, такі як масове створення замовлень та частин замовлень відповідно, масове зчитування сутностей з БД, масове оновлення даних сутностей а також запит з використанням JOIN [17];
- в сутностях мають бути налаштовані головні та зовнішні ключі (у БД), а також має бути прописан зв’язок між сутностями у java-класах.

Було прийнято рішення дотримуватись загально-прийнятої структури побудування проекту:

- взаємодіяти з БД можуть лише ті класи, що реалізують інтерфейси, іменовані репозиторіями, зазвичай такі репозиторії мають паттерн назви – «СутністьRepository», а реалізації названі – «СутністьDao»;
- в цей час також існують сервіси, що мають екземпляр класу репозиторію і звертаються до його методів, щоб отримати необхідні дані, або зробити запит до БД;
- обробка виняткових ситуацій;
- виведення повідомлень у разі виникнення помилок;

Таким чином, з головного класу є змогу викликати сервіс, який в свою чергу, викличе метод потрібного репозиторію, та поверне значення, якщо воно було. Така структура приближає проект до стану реального проекту.

Для коректного функціонування кінцевого продукту необхідними технічними умовами є:

- операційна система Windows 10, Linux, MacOS;
- наявність PostgreSQL-клієнту;
- обсяг оперативної пам’яті (ОЗУ) не менш 3 ГБ.

Java – мова програмування, яка була обрана для розробки ПЗ. Код повинен бути написаний за всіма стандартами та згідно з договором про стиль коду мови Java. Для розробки програмного застосунку було використано середу розробки IntellijIDEA та фреймворк для автоматизації збору проекту на основі опису їх структури у файлі pom.xml.

У якості постачальника БД було обрано PostgreSQL [11] – об’єктно-реляційна СУБД (Система Управління Базами Даних), що дозволяє використовувати як традиційні табличні структури, так і об'єктно-орієнтовані концепції. Це означає, що можна використовувати переваги звичайних реляційних таблиць та водночас працювати зі складними структурами даних.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1. Функціональне призначення програми

Результатом виконання цієї кваліфікаційної роботи має бути програмний додаток, який буде мати на меті порівняння технологій взаємодії з БД. Додаток має відправляти запити до БД та повертати необхідну відповідь, а також мусять бути відповідні методи для аналізу та порівняння.

2.2. Використана архітектура проекту

У розгляді архітектури розробленого Java-застосунку для аналізу технологій та методів взаємодії з базами даних, було обрано модель, що відповідає принципам об'єктно-орієнтованого програмування (ООП) та включає в себе використання паттернів проектування.

ООП Принципи: Архітектура базується на основних принципах об'єктно-орієнтованого програмування, таких як спадкування, інкапсуляція та поліморфізм. Це дозволяє створювати модульний та розширюваний код, об'єднуючи пов'язані функціональність у класи та об'єкти.

Використання Паттернів Проектування: У проекті використані різні паттерни проектування, такі як Singleton для забезпечення єдиності екземпляра певного класу.

Модульність та Компонентний Підхід: Додаток розділений на модулі та компоненти, кожен з яких відповідає за конкретний аспект функціональності. Це сприяє зручній розробці, тестуванню та підтримці.

Використання JDBC та Hibernate: Зверху на архітектурному шарі знаходиться взаємодія з базою даних, яка вирішена використанням Java Database Connectivity (JDBC) для прямого з'єднання та Hibernate для об'єктно-реляційного відображення та спрощення роботи з базою даних.

Інтеграція з Іншими Технологіями: Архітектура передбачає можливість інтеграції з іншими технологіями, такими як Logback [15] для логування та HikariCP [14][18] для ефективного пулінгу з'єднань, що підвищує продуктивність та стабільність.

Всі ці складові узгоджено працюють разом, створюючи гнучку та розширювану архітектуру, яка відповідає вимогам розробки Java-застосунку для взаємодії з базами даних.

2.3. Використані технології та мова програмування

Дане ПЗ було розроблено з використанням наступних технологій, бібліотек, що були завантажені для роботи проекту через залежності у pom-файлі:

- Java
- Maven
- PostgreSQL
- Hibernate
- Lombok
- Logback
- HikariCP
- Hibernate-hikariCP

Java (рис. 2.1.) — це мова програмування, що є об'єктно-орієнтованою. У офіційній реалізації Java-програми компілюються у byte-код, що в свою чергу при виконанні інтерпретується віртуальною машиною для конкретної платформи операційної системи. Використовування даної мови програмування не залежить від ОС, Java є усюди, в цьому велика перевага.

Важливою особливістю мови Java є те, що його код спочатку транслюється в спеціальний byte-код, що підтримується різними платформами, а потім він переходить під опіку віртуальної машини JVM (Java Virtual Machine). Java не є чисто компільованою мовою, як С або С ++, але в цей же час відрізняється й від

таких мов, як Python або Ruby, різниця в тому, що ці мови не компілюють свій код, він одразу виконується інтерпретатором.

У мови Java є слоган, звучить він наступним чином – «Write Once, Run Anywhere», це означає, що програму, написану Java можна запускати на будь-якій ОС (Linux, Windows, Mac і т.д.) без компіляції, це забезпечує крос-платформенність і апаратну переносимість програм на Java. Хоч для кожної з платформ може бути своя реалізація віртуальної машини JVM, але кожна з них, без усіляких винятків, може виконувати один і той самий код.

Також, ще однією особливістю Java є те, що вона підтримує автоматичну збірку сміття(GarbageCollector), отже програмісту не треба власноруч звільняти пам'ять від об'єктів, що вже втратили своє посилання, а отже більше не будуть використовуватись, збирач сміття це зробить автоматично за вас, але все ж, згідно зі своєю логікою обробки. Наприклад, у C++ такої можливості немає.

Рис. 2.1. Логотип Java

Плюси:

- Принцип проектування ПЗ – ООП;
- Легкий для розуміння синтаксис, більше схожий на людську мову, аніж на машинний код;
- Сувора типізація даних надає більшу надійність коду;
- Крос-платформенність - Write once, run anywhere;
- Мультифункціональність;
- Автоматична збірка сміття(неактивні об’єкти або ті об’єкти, що втратили своє посилання, будуть видалені із пам’яті, що збереже ресурси на роботу решти програми);

- Ком’юніті, що постійно розвивається та допоможе на форумах;
- Потужний інструментарій для розробки Android-застосунків.

Мінуси:

- Велике навантаження на ОЗУ;
- Платне комерційне використання;
- Швидкість порівняно менша, ніж, наприклад, у C та C++.

Maven [10](рис. 2.2.) – це засіб автоматизації роботи з програмними проектами, для Java проектів. Створений для управління (management) та зборки (build) програм. За принципами роботи значно відрізняється від Apache Ant, та має простіший архітектурний вигляд щодо build-налаштувань, написаний в форматі XML. Даний файл описує проект, його зв'язки з зовнішніми модулями і компонентами, порядок будування (build), папки, необхідні зовнішні бібліотеки для розширення функціоналу та необхідні плагіни. Сервер із додатковими модулями та додатковими бібліотеками розміщується на серверах, звідки є відкритий доступ для завантаження і використання у своєму проекті.

Рис. 2.2. Логотип Maven

Структура проекту:

- src/main/java – директорія Java класів;
- src/main/resources – директорія конфігураційних файлів;
- src/test/java – директорія тестів.

PostgreSQL (рис. 2.3.) впроваджує об'єктно-реляційну модель, що дозволяє використовувати як традиційні SQL-запити, так і роботу зі складними об'єктами та структурами даних. Це забезпечує гнучкість та високий рівень абстракції під час взаємодії з базою даних. Також гарантує високий рівень надійності даних завдяки підтримці принципів ACID. Це робить її відмінним вибором для

застосунків, де стійкість та консистентність даних є критичними. PostgreSQL володіє розширеною підтримкою геопросторових типів даних, що робить її ідеальним вибором для застосунків, які працюють з географічними даними. Крім того, СУБД підтримує різноманітні типи даних, що розширюють її можливості у роботі з різнорідними даними. PostgreSQL користується підтримкою великої та активної спільноти розробників. Це гарантує постійне вдосконалення та вчасну підтримку, а також надає можливість швидко отримати поради від інших експертів у випадку виникнення проблем. Що найважливіше, PostgreSQL є вільним та відкритим програмним забезпеченням, що надає розробникам широкі можливості власних модифікацій та розширень. Це важливо для врахування унікальних вимог проекту.

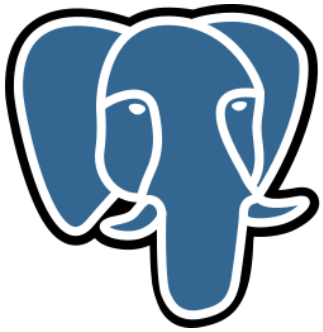

Рис. 2.3. Логотип PostgreSQL

Hibernate (рис. 2.4.), як об'єктно-реляційна система управління базами даних (ORM), надає зручний та високорівневий спосіб взаємодії з базами даних в середовищі розробки Java. Використання Hibernate дозволяє розробникам працювати з об'єктами у своєму коді, уникаючи необхідності написання складних SQL-запитів та забезпечуючи вищий рівень абстракції при роботі з даними.

Hibernate пропонує:

- Hibernate дозволяє розробникам працювати з об'єктами в їхньому коді, як звичайними Java-об'єктами, тим самим забезпечуючи більш природний та зрозумілий спосіб представлення даних.

- Hibernate може працювати з різними базами даних, що робить його універсальним інструментом для взаємодії з різноманітними системами управління базами даних.
- Hibernate автоматично генерує SQL-запити, що значно спрощує роботу розробників та дозволяє уникнути дублювання коду.
- Hibernate підтримує різні стратегії кешування, що сприяє підвищенню продуктивності за рахунок зменшення кількості запитів до бази даних.
- Hibernate автоматично управляє транзакціями, забезпечуючи атомарність, консистентність, ізоляцію та стійкість (ACID) при роботі з даними.
- За допомогою анотацій чи файлів маппінгу XML, Hibernate здійснює маппінг об'єктів на відповідні таблиці бази даних.

Hibernate є потужним інструментом для розробки Java-застосунків, що взаємодіють з базами даних, та відкриває нові можливості для ефективного та елегантного доступу до даних в об'єктно-орієнтованих проектах.

Рис. 2.4. Логотип Hibernate

Lombok [12](рис. 2.5.) – це бібліотека для мови програмування Java, яка спрощує розробку за допомогою автоматичного створення коду за допомогою аннотацій. Вона призначена для вирішення проблеми зайвого та повторюваного коду, що часто виникає при розробці на Java.

Основні Особливості Lombok:

- Lombok дозволяє вам застосовувати аннотації до класів, які автоматично генерують стандартні методи, такі як toString(), equals(), hashCode(), а також геттери та сеттери для поля.

- З використанням Lombok код може стати більш конкретним та зрозумілим, оскільки зменшується кількість необхідних ліній коду для створення базових елементів.
- Lombok включає аннотації, такі як @Data та @Builder, які автоматично створюють геттери, сеттери, метод toString(), метод hashCode(), метод equals(), а також шаблон Builder для покращення роботи з об'єктами.
- Аннотація @NonNull дозволяє визначити, що поле не може бути null, що полегшує роботу з винятками, пов'язаними з null-значеннями.
- Lombok надає аннотації, такі як @Slf4j або @Log, що автоматично генерують код для логування, що полегшує використання систем логування, таких як SLF4J.
- Lombok дозволяє легко створювати immutable-класи за допомогою аннотації @Value, що автоматично генерує код для невизначених полів та методів.

Lombok використовується для полегшення процесу розробки, зменшення кількості повторюваного коду та покращення читабельності та обслуговуваності коду в проектах на мові програмування Java. Бібліотека отримала назву на честь індонезійського острова Ломбок, розташованого неподалік від острова Ява. У перекладі з індонезійського Lombok означає «перець чилі»: за аналогією з приправою, бібліотека покликана підвищити якість Java-коду.

Рис. 2.5. Логотип Lombok

Logback (рис. 2.6.) – це потужний та гнучкий фреймворк для логування в мові програмування Java. Розроблений як покращений наступник фреймворку log4j, Logback надає розробникам широкий спектр можливостей для ефективного контролю та обробки логів у програмному забезпеченні.

Основні Характеристики Logback:

- Logback використовує простий логінговий фронтенд, такий як Simple Logging Facade for Java (SLF4J), що дозволяє використовувати різні реалізації логування, включаючи Logback.
- Центральною особливістю Logback є можливість гнучкого конфігурування через файли XML чи гнучку програмну конфігурацію. Це дозволяє налаштовувати різні аспекти логування, такі як формат виводу, рівні логування та місце призначення логів.
- Logback відомий своєю високою продуктивністю та низьким навантаженням на систему, що дозволяє використовувати його в різноманітних проектах без великої витрати ресурсів.
- Logback підтримує логування подій та винятків, що дозволяє ефективно виводити інформацію про винятки та події у великих програмах.
- Logback може логувати повідомлення в різні місця призначення, такі як консоль, файли, різні рівні важливості, включаючи TRACE, DEBUG, INFO, WARN, та ERROR.
- З Logback можна легко фільтрувати та перенаправляти лог-події відповідно до заданих критеріїв, що дозволяє розробникам докладно контролювати вивід логів.

Logback є популярним та потужним інструментом для логування в Java-проектах, дозволяючи ефективно керувати та аналізувати лог-інформацію для покращення процесу розробки та обслуговування програмного забезпечення.

Рис. 2.6. Логотип Logback

HikariCP є високопродуктивним та легким пулером з'єднань для мови програмування Java. В основі його функціоналу лежить швидкість та оптимізація

для забезпечення ефективного управління та використання з'єднань з базою даних.

Основні Особливості HikariCP:

- HikariCP славиться своєю високою швидкістю та ефективністю у використанні ресурсів, завдяки чому він є одним з найшвидших пулерів з'єднань для Java.
- Автоматичне керування розміром пулу дозволяє HikariCP динамічно адаптуватися до навантаження на базу даних, автоматично збільшуючи або зменшуючи кількість активних з'єднань в залежності від потреб додатка.
- HikariCP підтримує багато джерел даних, таких як реляційні бази даних (наприклад, MySQL, PostgreSQL, Oracle), а також NoSQL-джерела даних.
- Легко інтегрується з конфігураційними файлами та іншими інструментами керування конфігурацією, що дозволяє гнучко налаштовувати параметри пулу з'єднань.
- HikariCP надає різні статистичні дані та метрики, що дозволяє вам моніторити та аналізувати використання пулу з'єднань у реальному часі.
- Інтеграція HikariCP у проекти Java є досить простою за допомогою додавання необхідної залежності та налаштування параметрів пулу з'єднань.

HikariCP є відмінним вибором для забезпечення ефективного та оптимального використання з'єднань з базою даних у ваших Java-застосунках, особливо в умовах великих навантажень та вимог до швидкості.

Hibernate і HikariCP є двома популярними технологіями у своїх відповідних областях - ORM та пулінг з'єднань. Їх комбінація дозволяє створювати потужні та ефективні Java-застосунки для роботи з базами даних.

Основні Переваги Hibernate з HikariCP:

- Hibernate надає високорівневий механізм роботи з базою даних, а HikariCP додає до цього високоефективний пулер з'єднань, що сприяє збільшенню швидкодії взаємодії з базою даних.

- Обидві технології дозволяють гнучко налаштовувати свої параметри. Hibernate дозволяє вибирати стратегії отримання та збереження даних, в той час як HikariCP дозволяє контролювати розмір пулу з'єднань та інші аспекти пулінгу.
- HikariCP взаємодіє з Hibernate, автоматично керуючи з'єднаннями до бази даних. Це дозволяє оптимізувати використання ресурсів та забезпечити ефективне використання пулу з'єднань.
- Hibernate забезпечує роботу з транзакціями у високорівневому стилі, дозволяючи забезпечити атомарність операцій. HikariCP в свою чергу гарантує ефективне управління з'єднаннями під час виконання транзакцій.
- Як Hibernate, так і HikariCP, підтримують різні системи управління базами даних, роблячи їх універсальними інструментами для розробки.
- Обидві технології надають можливості для логування та моніторингу, що дозволяє розробникам отримувати важливу інформацію щодо використання та продуктивності пулу з'єднань.

Об'єднання Hibernate з HikariCP надає розробникам надійний та ефективний стек технологій для роботи з базами даних у Java-застосунках, забезпечуючи високу продуктивність та гнучкість.

2.4. Структура програми та алгоритми її функціонування

Отже, у даному розробленому проекті я використовував архітектуру більш-менш реального проекту[1], однак, по-перше, я створив два окремі модулі jdbc та hibernate, у них відповідно будуть міститися відповідні класи для реалізації завдання (рис. 2.7.). Наступним кроком я створив усі модулі всередині[2][3], для jdbc це: config, dao, model, repository, service; а для hibernate: dao, model, repository, service, util. Усі методи для тестування були зазначені у класах, що знаходились у корені модулей jdbc та hibernate, називались відповідно AppJDBC та AppHibernate. Також мушу зазначити, що у модулі resources є файли hibernate.cfg.xml та logback.xml. Перший присутній там для конфігурації hibernate для підключення до БД та інші налаштування, такі як пулінг

підключень до БД та маппінг моделей. Другий файл містить налаштування логів програми, це було додано для зручності інспекцій помилок коду.

Пройдемось по модулям, описаним вище:

- dao – модуль, в якому описані класи для прямого доступу до БД;
- model – модуль, в якому описані класи, що представляють собою сутності, описані у БД;
- repository – модуль, в якому описані інтерфейси для класів модуля dao;
- service – модуль, в якому описані класи-сервіси, що забезпечують доступ для користувача до БД через dao модуль;
- config – модуль, що містить клас-конфігурацію для jdbc-реалізації;
- util – модуль, що містить клас для створення сесії для hibernate, за налаштуваннями файлу hibernate.cfg.xml.

Таким чином, дана структура більше схоже на структуру реального проекту ІМ, що зберігає в собі принцип ізолювання даних від користувача. Класи різних модулей (мова про jdbc та hibernate) в більшості випадків схожі між собою, тому що мета була реалізувати технології таким чином, аби їх порівняння було більш прозорим та об’єктивним.

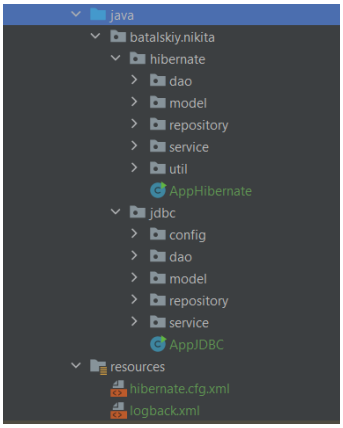

Рис. 2.7. Структура проекту

Так як у якості БД було обрано PG, то маємо наступну структуру: сервер graduate, в ньому вже бази даних jdbc та hibernate, дана БД так влаштована, що

далі йдуть схеми, в яких вже самі таблиці. Я використовував дефолтні схеми public. В кожній базі відповідно знаходяться сутності у якості таблиць: user, product, order, order_item.

Таблиця user відповідає за сутність користувача, має наступні стовпчики: id, name, lastname, address, email. Id – ідентифікатор, для таких стовпців налаштовують тип поля bigserial, у PG після збереження таблиці та її властивостей поле змінюється на bigint, проте додається послідовність, що записує кожну нову сутність з id на один більше, ніж попередній. У всіх сутностей він є, тож далі він буде тільки перелічуватись. Name – поле строки довжиною у 15 символів, що має відображати ім’я, lastname – теж саме, проте стосується прізвища. Address – це вже поле тексту, воно не має обмежень по довжині, email – теж поле тексту без явно зазначеної довжини. У цій таблиці налаштований тільки первинний ключ, зазвичай це стовпчик id (таб. 2.1.).

Таблиця 2.1

Структура об’єкту user

Назва колонки	Тип	Призначення
id	Bigint	Унікальний ідентифікатор користувача
name	Varchar	Ім’я користувача, має бути унікальною
lastname	Varchar	Прізвище користувача, має бути унікальним
address	Text	Адреса користувача, має бути унікальна
email	Text	Пошта користувача, має бути унікальною

Таблиця product відповідає за сутність продукту, має наступні стовпчики: id, name, description, price. Name – поле тексту без обмежень довжини, так як

важко визначити наперед довжину назви продукту, description – теж текст без обмежень довжини по тим самим причинам, що й минулий стовпець. Price – поле ціни, було зазначено у якості цілого числа для зручності. У цій таблиці налаштований теж тільки первинний ключ - стовпчик id (таб. 2.2.).

Таблиця 2.2

Структура об’єкту product

Назва колонки	Тип	Призначення
id	Bigint	Унікальний ідентифікатор продукту
name	Text	Назва продукту
description	Text	Більш детальній опис продукту
price	Int	Ціна продукту

Таблиця order відповідає за сутність замовлення, має наступні стовпчики: id, user_id, date, status. User_id – поле цілого числа, яке посилається на id сутності користувача, який створив це замовлення. Date – поле типу дата, що покликано відображати дату створення замовлення, воно фіксується під час створення сутності за допомогою програмного коду. Status – поле строки довжиною 15 символів, передбачається використання статусів як «Новий», «Обробляється» і так далі. У цій таблиці налаштований первинний ключ - стовпчик id, а також зовнішній – що посилається на поле id у таблиці user (таб. 2.3.).

Таблиця 2.3

Структура об’єкту orders

Назва колонки	Тип	Призначення
id	Bigint	Унікальний ідентифікатор замовлення

user_id	Bigint	Унікальний ідентифікатор користувача, що створив це замовлення. Зовнішній ключ.
date	Date	Дата та час створення замовлення
status	Varchar	Статус замовлення

Таблиця order_item відповідає за сутність частини замовлення, має наступні стовпчики: id, order_id, product_id, quantity. Order_id – поле цілого числа, яке посилається на id сутності замовлення, до якого воно відноситься. Product_id – поле цілого числа, яке посилається на id сутності продукту, який є частиною цього замовлення. Quantity – поле цілого числа, що вказує кількість даного продукту. У цій таблиці налаштований первинний ключ - стовпчик id, а також зовнішні: order_id – що посилається на поле id у таблиці order та product_id – поле id у таблиці product (таб. 2.4.).

Таблиця 2.4

Структура об’єкту order_item

Назва колонки	Тип	Призначення
id	Bigint	Унікальний ідентифікатор замовлення
order_id	Bigint	Унікальний ідентифікатор замовлення. Зовнішній ключ.
product_id	Bigint	Унікальний ідентифікатор продукту, що потрапив у замовлення. Зовнішній ключ.
quantity	Numeric	Кількість сигарет

Діаграма таблиць у БД виглядає наступним чином, вона ілюструє зв’язок між таблицями та надає графічний вигляд усьому описаному вище тексту (рис. 2.8.).

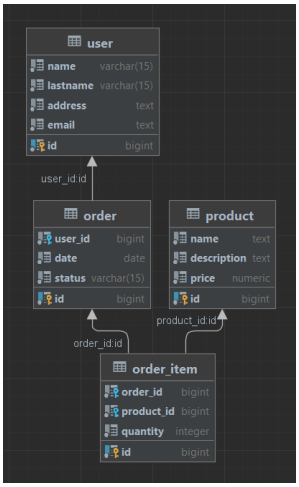

Рис. 2.8. Діаграма таблиць сутностей у БД

Тепер пройдемось більш детально по кожному з модулей, почнемо з модулей repository (рис. 2.9.). Загалом, різниці між класами у jdbc та hibernate реалізаціях[4][5] у класах цього модуля немає, тож роздивимось приклади з однієї з реалізацій (рис. 2.10.). Усі репозиторії являють собою інтерфейси, що лише задають необхідний напрямок. В них написані усі потрібні функції, включаючи стандартні CRUD-операції.

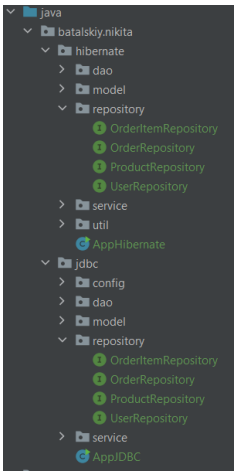

Рис. 2.9. Модулі repository

Рис. 2.10. Репозиторії

Наступними роздивимось модулі dao[7][8] (рис. 2.11.). Тут вже написані класи, що реалізують зазначені вище інтерфейси та їх функції. Розглянемо на прикладі OrderDao класу з jdbc та hibernate частини (рис. 2.12., 2.13.). Методи однакові, проте через використання різних технологій для обробки запитів через БД – реалізації методів відрізняються.

- getOrderById(long id) – метод, що повертає сутність, шукаючи її за id по БД. У jdbc реалізації викликається sql-запит «SELECT * FROM jdbc.public.order WHERE id = ?», даний запит повертає ResultSet, що містить усі стовпці таблиці jdbc.public.order, де стовпчик id співпадає з переданим у метод.

В свій час у hibernate метод кардинально відрізняється, нам взагалі не потрібно писати sql-запит, просто потрібно використати метод сесії session.get(Order.class, orderId), він поверне дані потрібної сутності і сам створить java-об’єкт (завдяки анотаціям у класах моделях.

- getOrdersByUserId(long userId) – метод для отримання замовлення по id користувача. У jdbc викликається sql-запит «SELECT * FROM jdbc.public.order WHERE id = ?», потребує додаткового налаштування запиту, щоб передати потрібний id. У hibernate «FROM Order o WHERE o.user.id = :user_id», теж потребує налаштування id.
- getAllOrders() – метод, що повертає усі сутності таблиці без винятків. У jdbc йде виклик sql-запиту «SELECT * FROM jdbc.public.order», що повертає ResultSet, з якого ми дістаємо та записуємо наші сутності. У hibernate знову код виглядає простіше - «FROM Order», та додатково передається клас у програмі, що відображає потрібну сутність - Order.class, хоча під капотом він визиває такий же запит, як і jdbc.
- addOrder(Order order) – метод для запису об’єкту у БД, він нічого не повертає. Jdbc – «INSERT INTO jdbc.public.order (user_id, date, status) VALUES (?,?,?)"», потім потрібно ще налаштувати запит, щоб передати потрібні поля об’єкту. Hibernate – session.merge(order), де order – потрібний нам об’єкт у застосунку.
- updateOrder(Order order) – метод для оновлення даних сутності у БД, нічого не повертає. Jdbc – «UPDATE jdbc.public.order SET status = ? WHERE id = ?», потребує додаткового налаштування запиту, щоб назначити потрібний id. Hibernate – потрібно створити тимчасовий об’єкт сутності, щоб в ньому змінити потрібні поля, а потім відправити його до БД методом session.merge(orderToUpdate).
- deleteOrder(long id) – метод для видалення сутності з БД, нічого не повертається. Jdbc – «DELETE FROM jdbc.public.order WHERE id = ?», знову потрібні налаштування, щоб вказати, який саме об’єкт повинен бути видалений.

Hibernate – спочатку використання методу для отримання потрібної сутності з БД – session.get(Order.class, orderId), потім вже видалення його – session.remove(orderToDelete).

- getOrdersWithDetails(long userId)[9] – метод, що був розроблений для пошуку з використанням функції БД JOIN, в моїй розробці повертається список строк, хоча це й не принципово, бо даний метод був розроблений тільки для тестування. Jdbc – «SELECT o.id, o.user_id, o.date, o.status, p.name, p.price "
  + "FROM jdbc.public.order o "
  + "JOIN jdbc.public.order_item i ON o.id = i.order_id "
  + "JOIN jdbc.public.product p ON i.product_id = p.id "
  + "WHERE o.user_id = ?», потреба в налаштуванні o.user_id у запиті. Hibernate – «SELECT o.id, o.user.id, o.date, o.status, p.name, p.price "
  + "FROM Order o "
  + "JOIN OrderItem i on i.order.id = o.id "
  + "JOIN i.product p "
  + "WHERE o.user.id = :userId», теж присутня потреба в налаштуванні параметру userId.
- getMaxOrderId – метод, що був розроблений задля пошуку максимального значення id у відповідній таблиці (потрібно для роботи методів для порівняння). У jdbc цей метод викликає sql-запит «SELECT id FROM jdbc.public.order ORDER BY id DESC LIMIT 1», фактично він дістає усі id з таблиці jdbc.public.order, сортує по зменшенню, тобто спочатку буде найвищий показник, і на останок обмежує відповідь усього лише однією строкою, так і отримуємо потрібне нам значення. У hibernate це все виглядає значно простіше, хоча всередині, скоріш за все, він робить такий самий sql-запит, проте у коді виглядає воно значно приємніше та зрозуміліше «SELECT MAX(o.id) FROM Order o», в нього налаштована функція MAX, що й повертає нам максимальне значення потрібної колонки у таблиці.
- mapOrder(ResultSet resultSet) – подібні функції присутні тільки у jdbc реалізаціях, бо потрібно привести результат запиту до БД у java-клас, тобто в

цьому методі ми за допомогою класу ResultSet отримуємо відповідь від БД на наш запит, та створюємо потрібний нам об’єкт в середовищі.

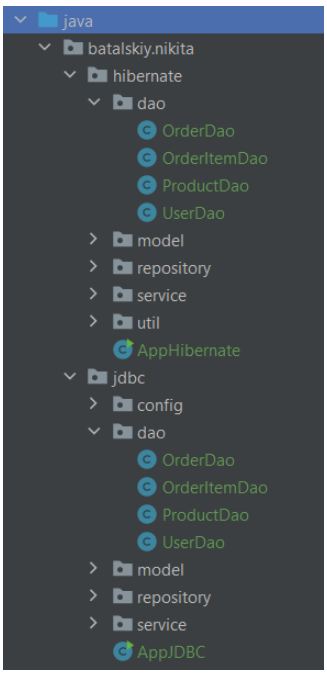

Рис. 2.11. Модулі dao

```
@Override
public void addOrder(Order order) {
  try (Connection connection = dataSource.getConnection()) {
    PreparedStatement stmt =
        connection.prepareStatement(
            sql: "INSERT INTO jdbc.public.order (user_id, date, status) VALUES (?,?,?)");
    stmt.setLong( parameterIndex: 1, order.getUser().getId());
    stmt.setObject( parameterIndex: 2, order.getDate());
    stmt.setString( parameterIndex: 3, order.getStatus());
    stmt.executeUpdate();
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}
```

```
@Override
public void updateOrder(Order order) {
  try (Connection connection = dataSource.getConnection()) {
    if (getOrderById(order.getId()) != null) {
      PreparedStatement stmt =
          connection.prepareStatement( sql: "UPDATE jdbc.public.order SET status = ? WHERE id = ?");
      stmt.setString( parameterIndex: 1, order.getStatus());
      stmt.setLong( parameterIndex: 2, order.getId());

      stmt.executeUpdate();
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}
```

Рис. 2.12. Приклад методів jdbc-реалізацій

```java
@Override
public void addOrder(Order order) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    session.merge(order);
    session.getTransaction().commit();
  }
}
```

```java
@Override
public void updateOrder(Order order) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    Order orderToUpdate = session.get(Order.class, order.getId());
    orderToUpdate.setUser(order.getUser());
    orderToUpdate.setStatus(order.getStatus());
    session.merge(orderToUpdate);
    session.getTransaction().commit();
  }
}
```

Рис. 2.13. Приклад методів hibernate-реалізації

Тепер розглянемо модуль model (рис. 2.14.), на прикладі класів OrderItem кожної з реалізацій (jdbc – рис. 2.15., hibernate – рис. 2.16.). Класи однієї реалізації загалом схожі між собою структурно. Таким чином у jdbc реалізації класи не мають ніяких анотацій над полями, бо jdbc просто не вміє їх відслідковувати, усі поля зазначені згідно сутності у БД, окрім того, що поля зовнішніх ключів вказані не як id, а як цілі сутності – це загально прийнятий принцип[20][21]. Також у класах присутні анотації від фреймворку Lombok, вони зменшують кількість коду у класах, наприклад, анотація @NoArgsConstructor – створює невидимий конструктор класу без параметрів, анотації @Getter та @Setter – таким же чином створюють методи get та set для полів класу. У hibernate реалізації існують такі ж анотації над класом сутності, проте додатково додається дві анотації - @Entity та @Table(name = "order_item", schema = "public") [19], де у параметр name ми передаємо назву таблиці у БД, що має відображати цей клас. Також додаються анотації до полів класу, поле id відзначається анотаціями @Id,@GeneratedValue(strategy = GenerationType.IDENTITY) та @Column (name = "id"). Анотація @Id вказує на

те, що дане поле є id сутності, @GeneratedValue(strategy = GenerationType.IDENTITY) вказує на те, що дане поле інкрементується автоматично, а @Column (name = "id") вказує на конкретний стовпчик сутності у таблиці БД. Додатково, поля, що відображають зовнішні ключі, анотовані наступним чином - @ManyToOne та @JoinColumn(name = "order_id"), перша анотація вказує на тип зв’язку між таблицями, а друга – до якого стовпчику відноситься анотоване поле класу.

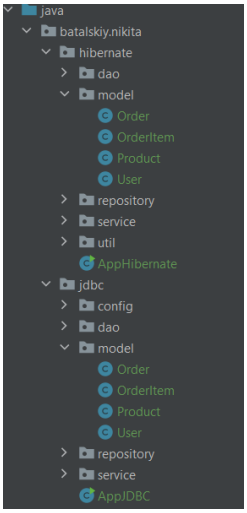

Рис. 2.14. Модулі model

```java
@Getter
@Setter
@EqualsAndHashCode
@ToString
@AllArgsConstructor
@NoArgsConstructor
public class OrderItem {
  private long id;
  private Order order;
  private Product product;
  private int quantity;

  new *
  public OrderItem(Order order, Product product, int quantity) {
    this.order = order;
    this.product = product;
    this.quantity = quantity;
  }
}
```

Рис. 2.15. Клас OrderItem jdbc реалізації

```java
@Entity
@Table(name = "order_item", schema = "public")
@Getter
@NoArgsConstructor
@EqualsAndHashCode
@ToString
public class OrderItem {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column (name = "id")
  private long id;

  @ManyToOne
  @JoinColumn(name = "order_id")
  private Order order;

  @ManyToOne
  @JoinColumn(name = "product_id")
  private Product product;

  @Column(name = "quantity")
  private int quantity;
```

Рис. 2.16. Клас OrderItem hibernate реалізації

Папка Service (рис. 2.17.) відповідає за бізнес-логіку застосунку, саме сюди звертається клієнт, а далі сервіс звертається до БД через екземпляри репозиторію та повертає клієнту отриману відповідь.

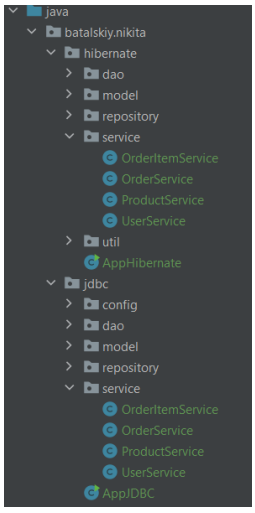

Рис. 2.17. Сервіси ПЗ

Загалом, класи сервісу мають ту саму ж структуру методів й самі методи, як і описані у класах dao модулю, бо вони власне викликають репозиторій та повертають теж саме, що й репозиторій після запиту до БД.

Роздивимось модулі конфігурації з'єднання з БД різними технологіями(jdbc – рис. 2.18. та hibernate – рис. 2.19.). В цих класах використовував зовнішню бібліотеку HikariCP [18] для налаштування пулу з'єднань. Для з'єднання java-застосунку з БД нам потрібні дані: адреса (у нашому випадку БД знаходиться на локальному комп'ютері, тому можна побачити localhost), ім'я користувача БД (використовував існуючого за замовчуванням користувача postgres) та пароль (при завантажені був встановлений пароль для цього користувача). У випадку jdbc реалізації, збираю всі потрібні налаштування у класі HikariConfig, а потім за ними створюю об'єкт класу HikariDataSource. У hibernate реалізації [22]все трохи інакше, тут нам потрібно на виході отримати об'єкт класу SessionFactory, а конфігуруємо його ми за допомогою файлу hibernate.cfg.xml (рис. 2.20.), власне, у ньому прописані всі потрібні дані для підключення до БД, а також додаткові налаштування, використовуючи пул з'єднань від бібліотеки HikariCP.

```
public class DatabaseConfig {
    1 usage
    public static final String URL = "jdbc:postgresql://localhost/jdbc";
    1 usage
    public static final String USERNAME = "postgres";
    1 usage
    public static final String PASSWORD = "653241";
    2 usages
    private static HikariDataSource dataSource;

    4 usages  new *
    public static DataSource getDataSource() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(URL);
        hikariConfig.setUsername(USERNAME);
        hikariConfig.setPassword(PASSWORD);
        hikariConfig.setMaximumPoolSize(50);
        dataSource = new HikariDataSource(hikariConfig);
        return dataSource;
    }
}
```

Рис. 2.18. Конфігурація з'єднання ПЗ з БД за допомогою jdbc

Рис. 2.19. Конфігурація з’єднання ПЗ з БД за допомогою hibernate

Рис. 2.20. Файл hibernate.cfg.xml

Наостанок розглянемо головні файли, да прописана логіка для порівняння технологій – AppJDBC та AppHibernate[23][24]. Знову ж таки, між ними різниця тільки у використаних технологій, а самі методи для порівняння було зроблені однаковими, аби все було більш об’єктивно. Головні методи класу (рис. 2.21.) теж виглядають однаково (різниця тільки у використаних екземплярах класів, хоч вони називаються однаково, проте відносяться кожний до своєї реалізації).

```java
public class AppHibernate {
    3 usages
    static UserService userService = new UserService();
    3 usages
    static ProductService productService = new ProductService();
    5 usages
    static OrderService orderService = new OrderService();
    5 usages
    static OrderItemService orderItemService = new OrderItemService();

    new *
    public static void main(String[] args) {
        for (int i = 1; i <= 50000; i++) {
            User user = createUser(i);
            userService.addUser(user);
        }

        for (int i = 1; i <= 50000; i++) {
            Product product = createProduct(i);
            productService.addProduct(product);
        }

        createOrdersAndOrderItemsForTest();
        testReadPerformance();
        testUpdatePerformance();
        testJoinPerformance();
    }
```

Рис. 2.21. Головний метод головного класу

Тож спочатку я заповнюю таблиці користувачів та продуктів за допомогою циклу, використовуючи допоміжні методи createUser та createProduct (рис. 2.22.). Ці методи існують для заповнення таблиць даними-пустишками, бо з ними взаємодіють інші таблиці.

```java
1 usage   new *
private static Product createProduct(int i) {
    Product product = new Product();
    product.setName("Product " + i);
    product.setDescription("Product description " + i);
    product.setPrice(i);
    return product;
}

usage   new *
private static User createUser(int i) {
    User user = new User();
    user.setName("Name " + i);
    user.setLastName("LastName " + i);
    user.setAddress("Address " + i);
    user.setEmail("Email " + i);
    return user;
}
```

Рис. 2.22. Методи для заповнення БД початковими даними

Далі йде метод createOrdersAndOrderItemsForTest (рис. 2.23.), якщо у минулих циклах ми просто створювали об’єкти та записували їх у БД, то в цьому методі ми вже створюємо об’єкти класів Order та OrderItem в тій самій кількості, проте застосовуючи при цьому 50 потоків для приближення коду до комерційного стану. Всередині потоків викликаються методи generateTestOrder та generateTestOrderItem (рис. 2.24.), що генерують потрібні об’єкти на основі раніше створених. При генерації використовуються випадково отримані користувачі, продукти та замовлення для повноцінного заповнення потрібних сутностей. Пошук випадкової сутності проходить наступним чином (рис. 2.25.): за допомогою методу findMaxId ми визначаємо максимальне значення id в таблиці сутності, далі, використавши метод стандартної бібліотеки Math.random, встановили верхню межу випадкового id отримане значення з методу findMaxId, а нижню – 0, потім здійснюється пошук сутності за отриманим випадковим id. Так само з усіма іншими сутностями.

```
private static void createOrdersAndOrderItemsForTest() {
  ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);

  long start = System.currentTimeMillis();

  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            Order order = generateTestOrder();
            orderService.addOrder(order);

            OrderItem item = generateTestOrderItem();
            orderItemService.addOrderItem(item);
            iterationCount++;
          }
        });
  }
  executor.shutdown();
  try {
    executor.awaitTermination( timeout: 5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for create: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

Рис. 2.23. Метод createOrdersAndOrderItemsForTest

Рис. 2.24. Методи для генерації замовлень та їх частин

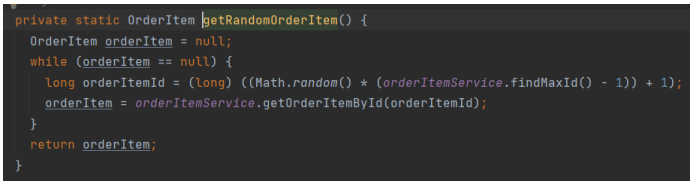

Рис. 2.25. Метод для пошуку випадкової сутності

Наступним роздивимось метод testReadPerformance (рис. 2.26.) для масового зчитування даних з БД. Тут нічого складного, ініціалізуємо десять потоків, кожному даємо завдання зробити тисячу зчитувань кожної сутності, для вибору сутності використовуємо вже знайомі нам методи для отримання випадкової сутності з БД.

Рис. 2.26. Метод для масового зчитування даних з БД

Далі роздивимось метод testUpdatePerformance (рис. 2.27.), який робить масове оновлення даних, у моєму випадку саме сутностей замовлення та його частин замовлення, де у замовлення змінюється статус (при створенні він був «New»), а частини замовлення випадковим чином змінюють кількість. Тут створюються 10 потоків, кожний з яких робить п’ять тисяч ітерацій, в яких дістає випадкове замовлення, змінює його статус, потім дістає усі частини замовлення та змінює кількість у кожного.

```
private static void testUpdatePerformance() {
  ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
  long start = System.currentTimeMillis();
  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            Order order = getRandomOrder();
            order.setStatus("In work");
            orderService.updateOrder(order);
            ArrayList<OrderItem> orderItemsByOrderId =
                (ArrayList<OrderItem>) orderItemService.getOrderItemsByOrderId(order.getId());
            orderItemsByOrderId.forEach(
                orderItem -> {
                  orderItem.setQuantity((int) (Math.random() * 9) + 1);
                  orderItemService.updateOrderItem(orderItem);
                });
            iterationCount++;
          }
        });
  }
  executor.shutdown();
  try {
    executor.awaitTermination( timeout: 5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for update: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}
```

Рис. 2.27. Метод для масового оновлення даних у БД

На останок розглянемо метод testJoinPerformance (рис. 2.28.), він використовує операнд JOIN у sql-запиті, докладніше було описано вище. Так само створюється десять потоків, які по п’ять тисяч разів викликають потрібний метод, передаючи випадкового користувача.

Рис. 2.28. Метод з використанням JOIN та зовнішніх ключів таблиць

Усі методи, що описані вище, огорнені простою логіку заміру часу на виконання, завдяки методам стандартної бібліотеки System замірюється час у мілісекундах перед початком виконання логіки методу, та після завершення роботи, їх різниця вираховується в кінці кожного методу та виводиться на екран.

2.5. Використані технічні та програмні засоби

В процесі тестування та розробки були використані наступні технічні засоби:

- оперативна пам’ять обсягом 16 гб;
- процесор Intel Core i5 11th Gen;
- накопичувач на SSD диску обсягом 1 тб;
- відеоадаптер NVIDIA GeForce RTX 3050 Ti;
- клавіатура та мишка.

Вказані технічні засоби не мають бути обов’язково ідентичних характеристик для безперешкодної роботи системи.

Під час розробки даного застосунку були використані такі програмні засоби:

- IntelliJ IDEA;
- Git, GitHub;

- PostgreSQL.

IntelliJ IDEA (рис. 2.29.) є інтегрованим середовищем розробки програмного забезпечення, призначеним для роботи з різними мовами програмування, зокрема Java, JavaScript, і Python, і розробленим компанією JetBrains. У версії Community підтримуються інструменти тестування TestNG і JUnit, системи контролю версій CVS, Subversion, Mercurial і Git, а також засоби складання Maven, Ant, Gradle. Мови програмування, такі як Java, Scala, Clojure, Groovy, Kotlin і Dart, також включені в підтримку. Серед інших функцій - підтримка розробки мобільних додатків для Android, модуль візуального проектування GUI-інтерфейсу Swing UI Designer, редактор регулярних виразів, XML-редактор, система перевірки коду, система контролю виконання завдань, а також додатки для імпорту та експорту проектів з Eclipse. Передбачені також інтеграційні засоби для систем відстеження помилок, таких як JIRA, Trac, Redmine, Pivotal Tracker, GitHub, YouTrack, Lighthouse.

Ultimate Edition - це комерційна версія, яка відрізняється наявністю підтримки додаткових мов програмування (PHP, Ruby, Python, JavaScript, CoffeeScript, HTML, CSS, SQL), підтримкою технологій Java EE, UML-діаграм, підрахунку покриття коду, можливістю роботи з фреймворками (Rails, Grails, Google Web Toolkit, Spring, Play Framework і Hibernate), а також інтеграцією з Perforce, Microsoft Team Foundation Server і Rational ClearCase.

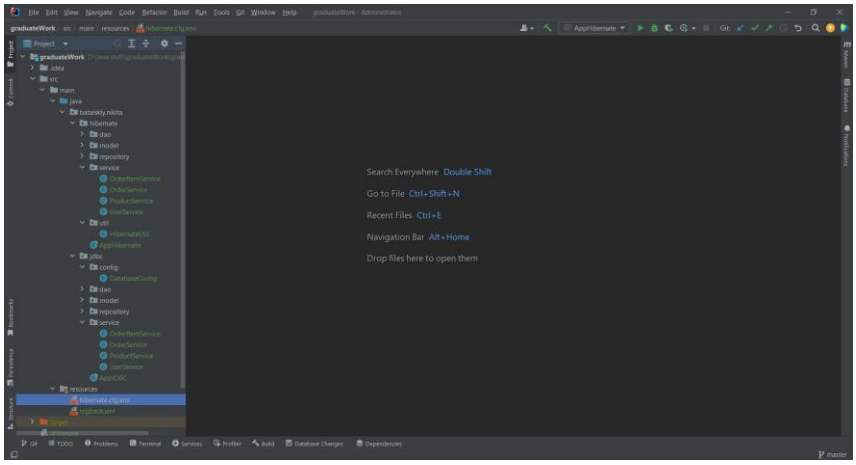

Рис. 2.29. Інтерфейс IntelliJ IDEA

Git представляє собою систему управління розподіленими версіями (DVCS), яка в основному використовується для розробки як відкритого, так і комерційного програмного забезпечення. DVCS забезпечують повний доступ до кожного файлу, гілки та ітерації проекту, даючи кожному користувачеві можливість переглядати повну та автономну історію всіх змін. На відміну від колишніх централізованих систем управління версіями, DVCS, такий як Git, не вимагає постійного з'єднання з центральним сховищем.

GitHub - це платформа для розміщення коду з можливістю контролю версій та співпраці. Вона дозволяє спільно працювати над проектами з будь-якого місця та зберігати версії проекту на віддаленому сервері.

PostgreSQL (рис. 2.33.) є потужною системою керування базами даних, яка надає розширені можливості для ефективного зберігання та обробки даних. Розроблена як відкрите програмне забезпечення, PostgreSQL володіє високим рівнем надійності та стабільності, забезпечуючи одночасно підтримку багатьох мов програмування та операційних систем.

У своїй основі PostgreSQL використовує об'єктно-реляційну модель даних, що дозволяє створювати складні структури даних та звіти. Це забезпечує високий рівень гнучкості та можливість використання різних типів даних, включаючи географічні дані, текстові документи та зображення.

PostgreSQL підтримує транзакційний підхід, що робить його ідеальним вибором для великих та складних додатків, де необхідна надійна обробка даних та забезпечення консистентності. Додатково, система включає розширені можливості індексації, що сприяють швидкому пошуку та фільтрації даних.

PostgreSQL також активно розвивається та оновлюється спільнотою розробників, що забезпечує постійне вдосконалення функціоналу та додавання нових можливостей. Завдяки своїй відкритості та гнучкості, PostgreSQL стає популярним вибором для різних проектів, від невеликих веб-додатків до великих корпоративних систем.

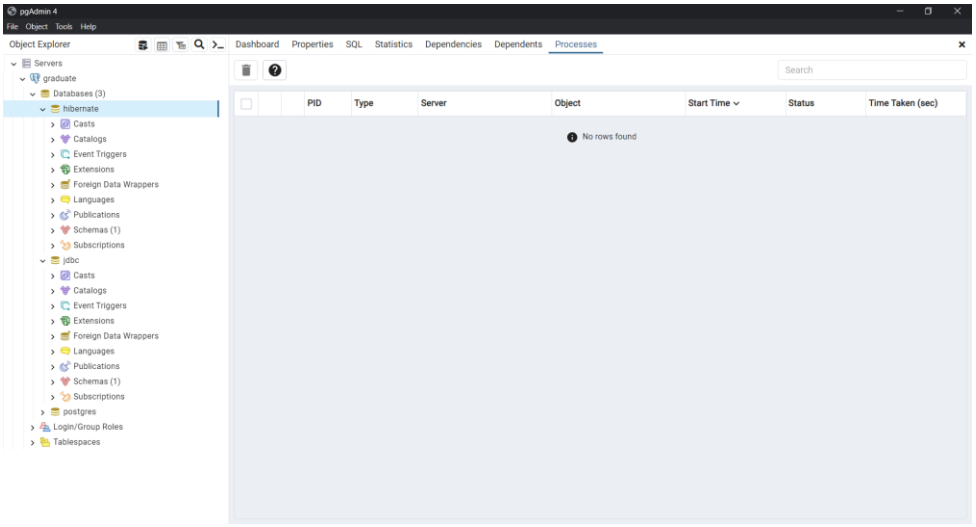

Рис. 2.30. Інтерфейс PostgreSQL

Для роботи з розробленим ПЗ локально потрібно мати встановлений PostgreSQL, у якому створити БД jdbc та hibernate, потім можна використати команду у консолі «psql -U postgres -d jdbc -f jdbc.sql» та «psql -U postgres -d hibernate -f hibernate.sql», де файли з розширенням .sql – це файли налаштування таблиць, що потрібні для роботи програми, зроблені за допомогою функції pg_dump [16], користувача можете використати свого, назвати самі бази даних теж можна по-іншому, але після цього потрібно буде зробити співвідносні зміни у файлах конфігурацій проекту. Далі просто запустити клас AppJDBC та AppHibernate.

РОЗДІЛ 3
ПОРІВНЯЛЬНИЙ АНАЛІЗ ТЕХНОЛОГІЙ

3.1. Результат першого запуску програми

Розпочнемо порівняння використаних технологій, для цього потрібно, щоб був запущений PG, потім – запустити головні класи по черзі. Аналіз проведено за допомогою налаштованих методів, усередині яких замірювався час на виконання методів, інтерфейс PG для статистики БД та jconsole для статистики використаних ресурсів.

Почнемо з jdbc реалізації, запустив клас, він почав викликати прописані методи, результати заміру часу можна побачити на рис. 3.1. Час на створення замовлень та частин замовлень, а також час на зчитування сутностей становив приблизно однаковий час – трохи більше шести секунд на виконання. Час, витрачений на оновлення даних у БД вже значно більший (тридцять три секунди), а час, витрачений на виконання операцій JOIN знадобився більше, ніж на оновлення (сорок дев’ять секунд).

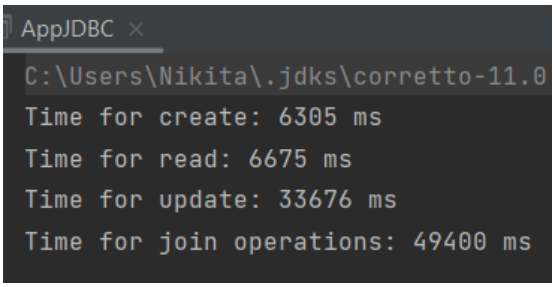

Рис. 3.1. Замір часу на виконання методів за допомогою jdbc

Тепер переглянемо статистику, що надає нам PG. За час виконання методів було створено близько ста сесій, з них активних усього близько п’ятнадцяти, коммітів – близько чотириста тисяч, кортежів запису – трохи більше за сімдесят п’ять тисяч, а оновлень – близько п’ятнадцяти тисяч, кортежів на повернення даних – близько восьми ста мільйонів (рис. 3.2.).

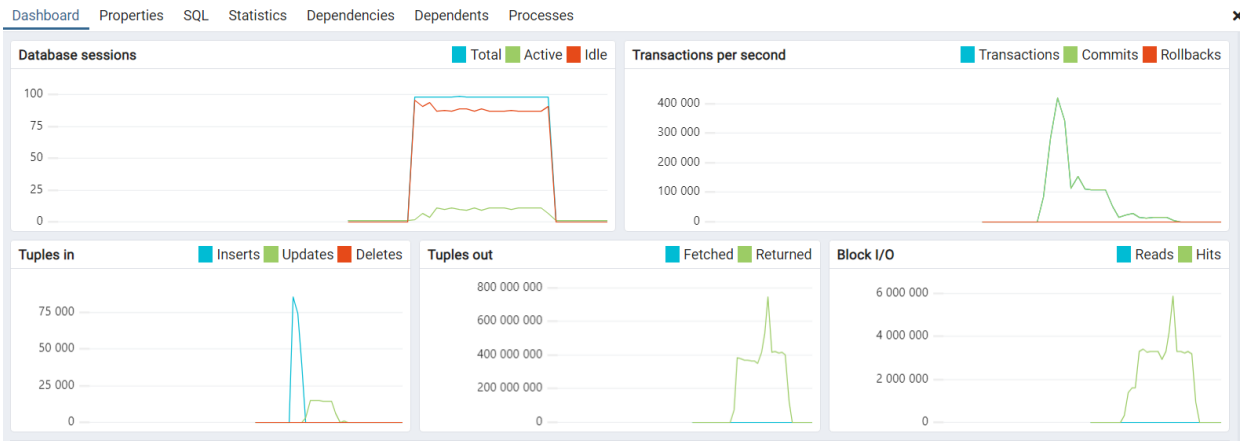

Рис. 3.2. Графік БД за час виконання методів

Перейдемо до статистики jconsole, вона надає статистику використаних ресурсів JVM, ми можемо побачити як графічний вигляд (рис. 3.3.), так і текстовий (рис. 3.4.). Можемо побачити, що за час роботи програми, декілька разів було використано сто п’ятдесят мегабайтів пам’яті так званої кучі, було створено усього сімдесят три потоки, з них живих тридцять дев’ять (у піку сорок три), усього класів було створено три тисячі сімсот тридцять два, а навантаження на CPU сягало двадцяти відсотків у піку.

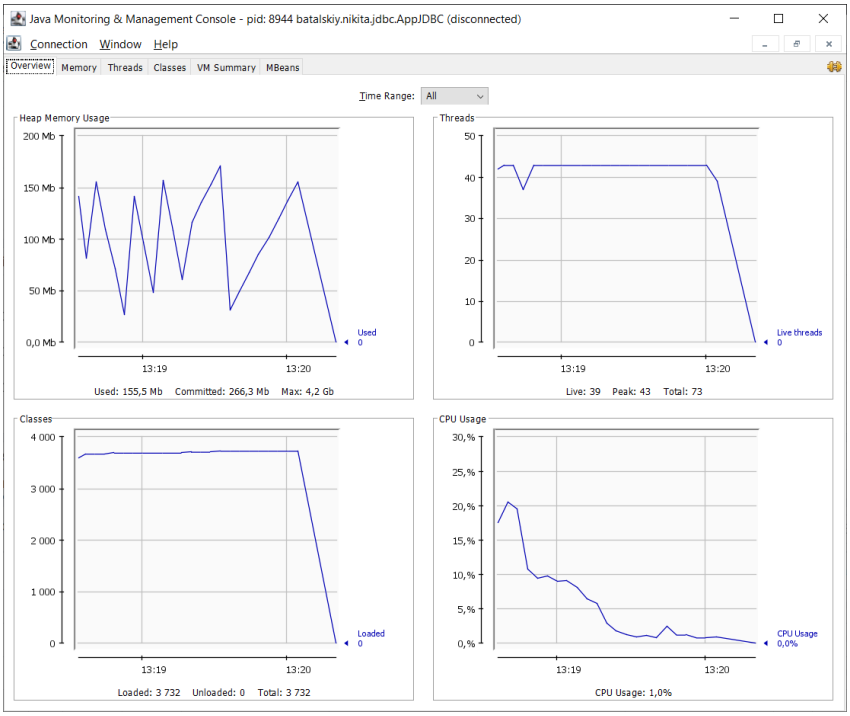

Рис. 3.3. Графічний вигляд статистики jconsole

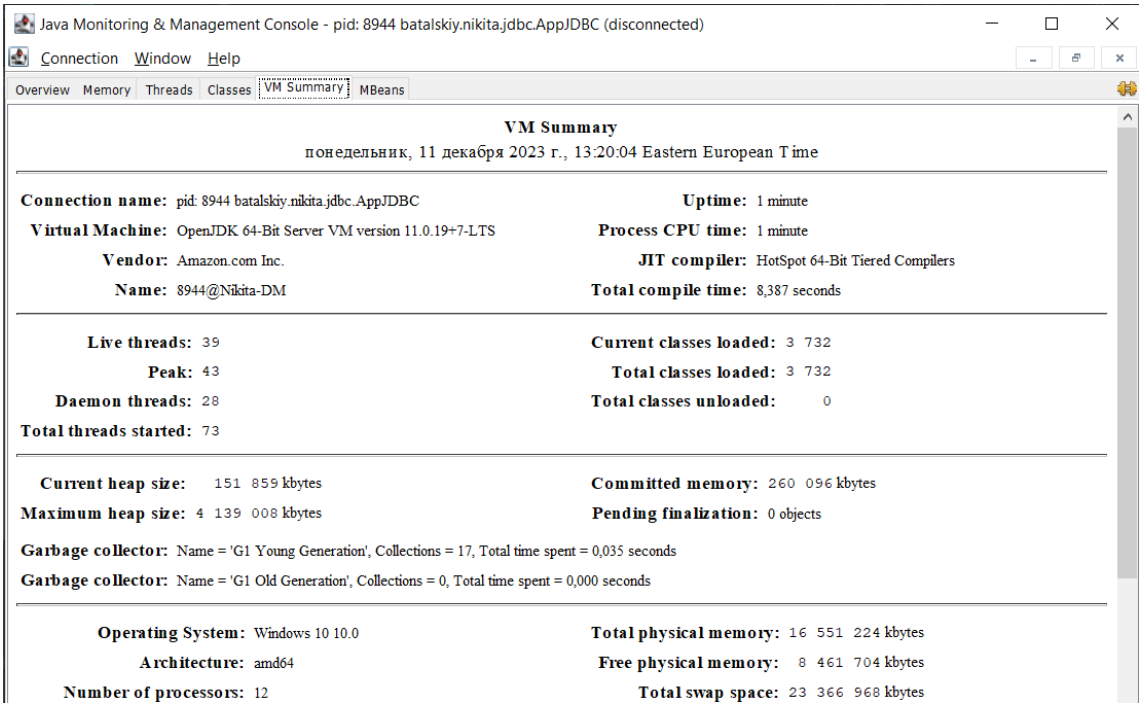

Рис. 3.4. Текстовий вигляд статистики jconsole

Тепер перейдемо до hibernate реалізації, алгоритм дій той самий. Час на створення замовлень та частин замовлень дорівнював одинадцять секунд, зчитування – сім секунд, оновлення даних – тридцять три секунди, а час на виконання операцій JOIN – п’ятдесят дві секунди (рис. 3.5.).

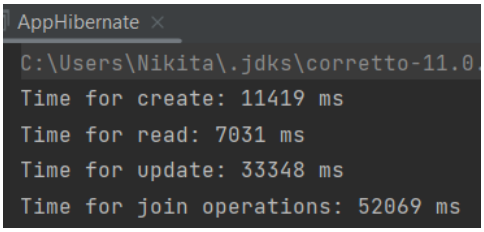

Рис. 3.5. Замір часу на виконання методів за допомогою hibernate

Переглядаючи статистику, що надає нам PG (рис. 3.6.), можна побачити, що сесій було близько п’ятдесяти усього, з них активних – приблизно п’ятнадцять, коммітів було у піку триста тисяч, кортежів на запис – приблизно п’ятдесят тисяч, на оновлення – близько п’ятнадцяти тисяч, на повернення даних – близько восьми ста тисяч.

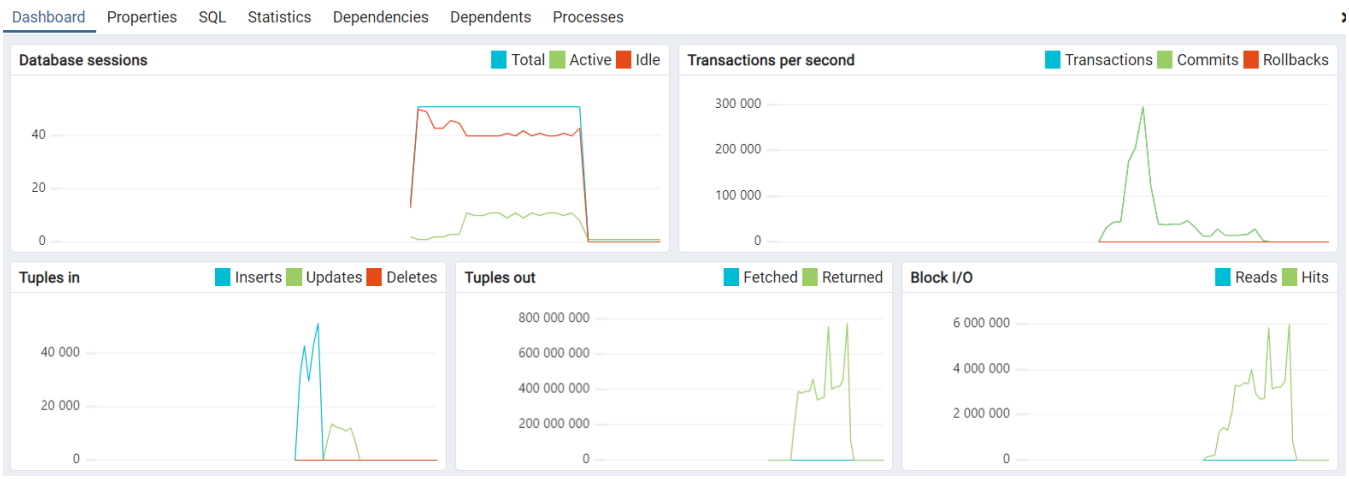

Рис. 3.6. Графік БД за час виконання методів

Далі подивимось на статистику, що надає нам застосунок jconsole (рис. 3.7. – 3.8.). Тут ми бачимо, що за час роботи програми, теж декілька разів було використано сто п’ятдесят мега байтів пам’яті кучі, усього потоків було створено сто сім, з них живих двадцять три (у піку двадцять шість), тим часом, класів було усього створено трохи більше десяти тисяч, а навантаження на CPU у піку становило сорок відсотків.

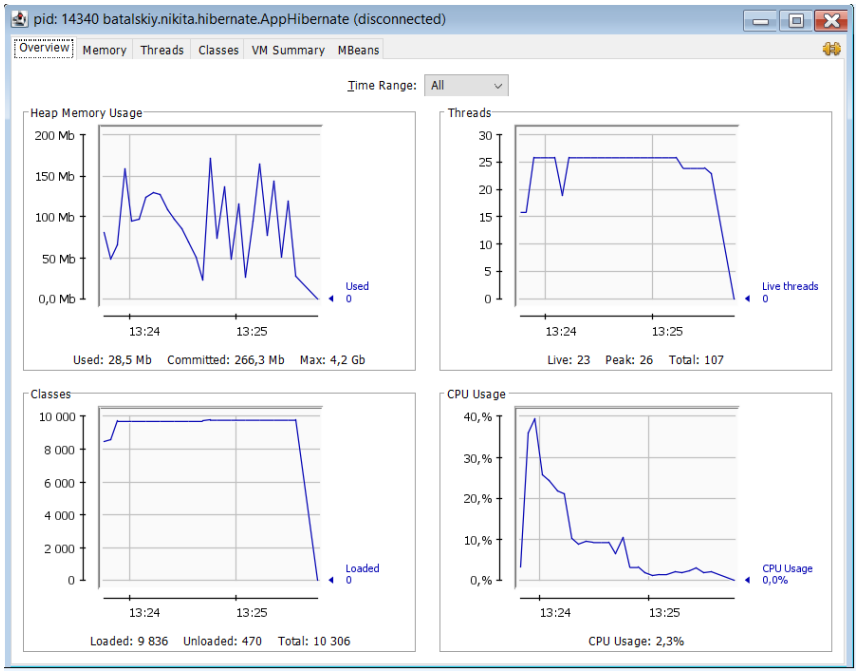

Рис. 3.7. Графічний вигляд статистики jconsole

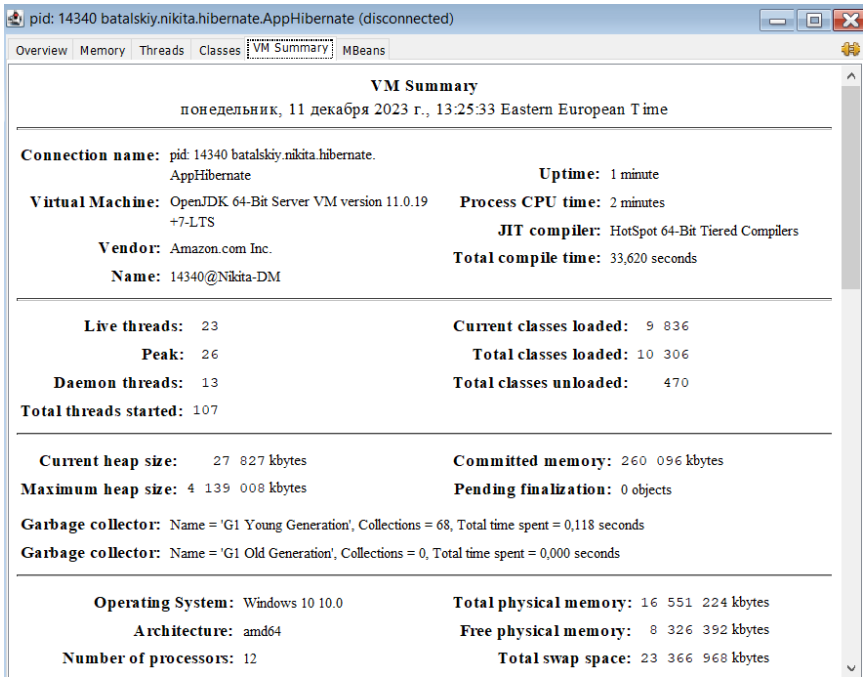

Рис. 3.8. Текстовий вигляд статистики jconsole

3.2. Результат другого запуску програми

Час (рис. 3.9.) на створення замовлень та частин замовлень, а також час на зчитування сутностей цього разу теж становив приблизно однаковий час, до того ж, майже такий самий – трохи більше шести секунд на виконання. Час, витрачений на оновлення даних у БД теж значно більший, порівняно з першими методами, проте приблизно такий самий, як і в перший запуск програми (тридцять три секунди), а час, витрачений на виконання операцій JOIN навіть трохи менше (сорок шість секунд).

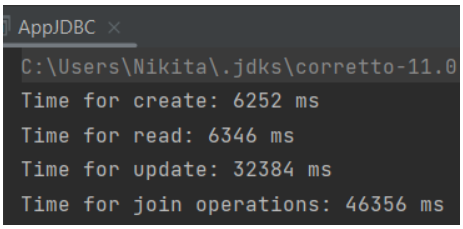

Рис. 3.9. Замір часу другого запуску jdbc

Щодо графіків з PG (рис. 3.10.), тут ситуація теж не сильно відрізняється. За час виконання створено близько ста сесій (активних – приблизно п’ятнадцять),

коммітів – як і в першому запуску, теж близько чотирьох ста тисяч, у цей же час, кортежів запису вийшло трохи більше – вісімдесят тисяч приблизно, кортежів оновлення приблизно стільки ж, як і в перший запуск, – близько п’ятнадцяти тисяч, кортежів на зчитування цього разу трохи менше – приблизно сімсот п’ятдесят тисяч.

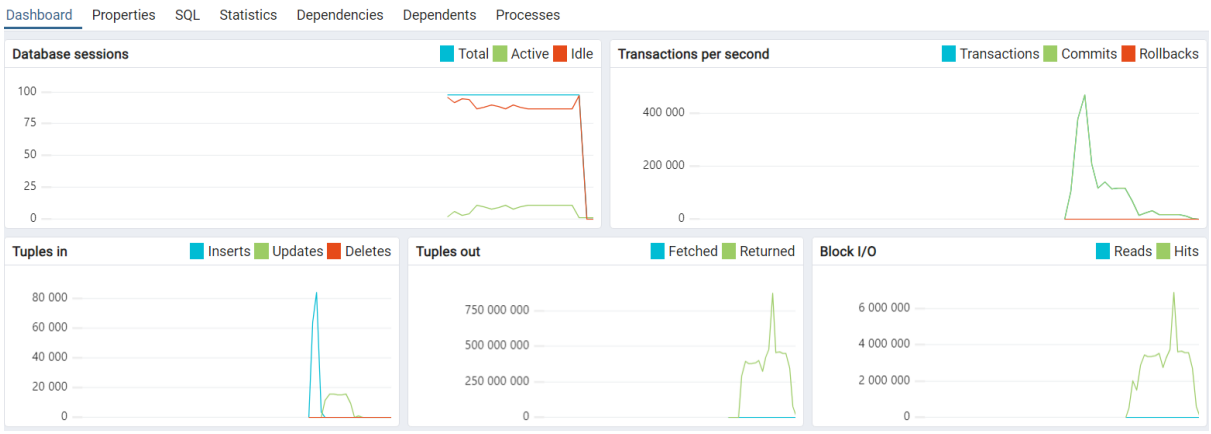

Рис. 3.10. Графік БД за час виконання методів

Щодо статистики jconsole (рис. 3.11., рис. 3.12.): загалом, ситуація ідентична першому запуску, теж декілька разів було зайнято сто п’ятдесят мега байтів кучі, близько сорока потоків створено, приблизно така сама кількість класів та навантаження на CPU.

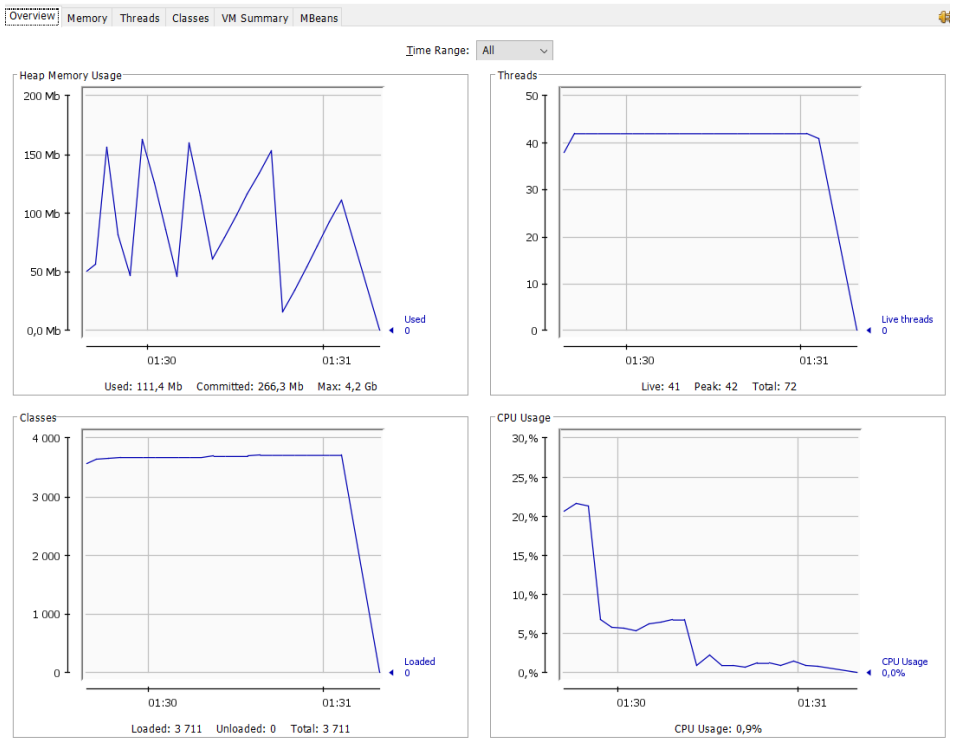

Рис. 3.11. Графічний вигляд статистики jconsole

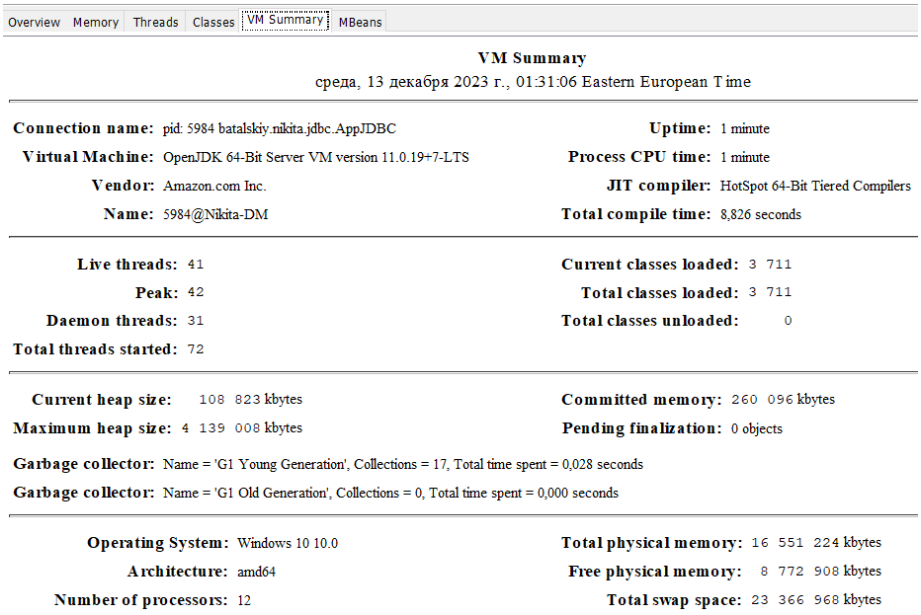

Рис. 3.12. Текстовий вигляд статистики jconsole

Тепер подивимось, що нам покаже hibernate за час своєї другої спроби. Щодо витраченого часу – загалом, він покращив свої результати (рис. 3.13.), але теж все приблизно однакове, час на запис – одинадцять секунд, зчитування – шість секунд, оновлення – тридцять дві секунди, JOIN – отут вже значно кращий результат, сорок шість секунд.

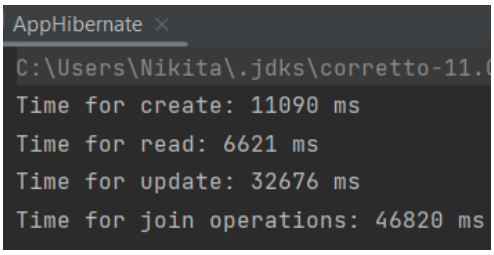

Рис. 3.13. Замір часу на виконання методів за допомогою hibernate

Дивлячись на графіки від PG (рис. 3.14.), можна сказати, що тут ситуація не змінилась: сесій було близько п’ятдесяти усього (активних – приблизно п’ятнадцять), коммітів було у піку триста тисяч, кортежів на запис – приблизно п’ятдесят тисяч, на оновлення – близько п’ятнадцяти тисяч, на повернення даних – близько восьми ста тисяч.

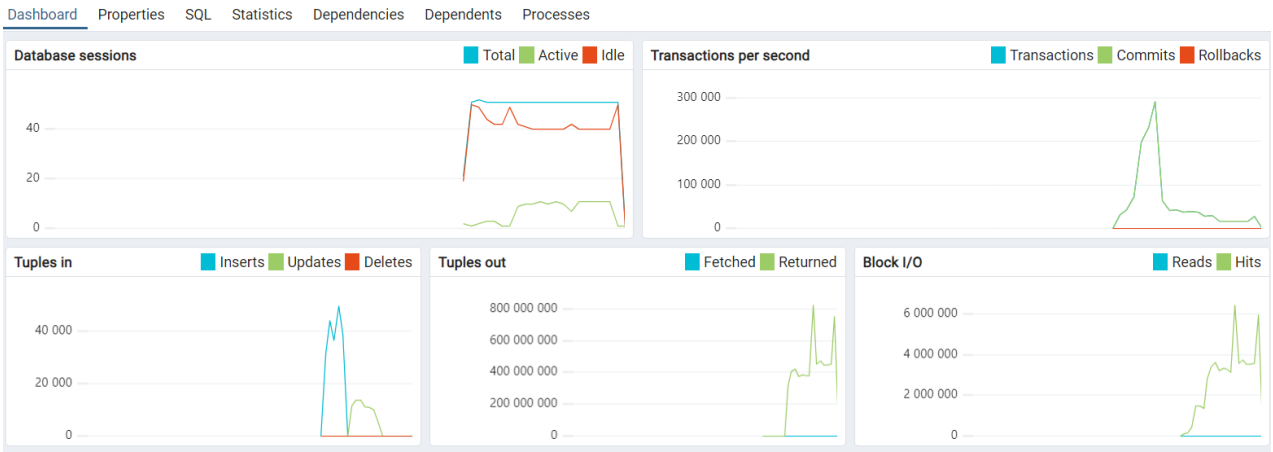

Рис. 3.14. Графік БД за час виконання методів

Перейдемо до jconsole (рис. 3.15., рис. 3.16.). Тут вже ситуація відрізняється від першого запуску, кучі було використано на майже двісті мега байтів кілька разів, проте з рештою все приблизно так само: близько двадцяти п’яти потоків, майже десять тисяч класів, цього разу навантаження на CPU сягнуло трошки більшого піку – п’ятдесят відсотків.

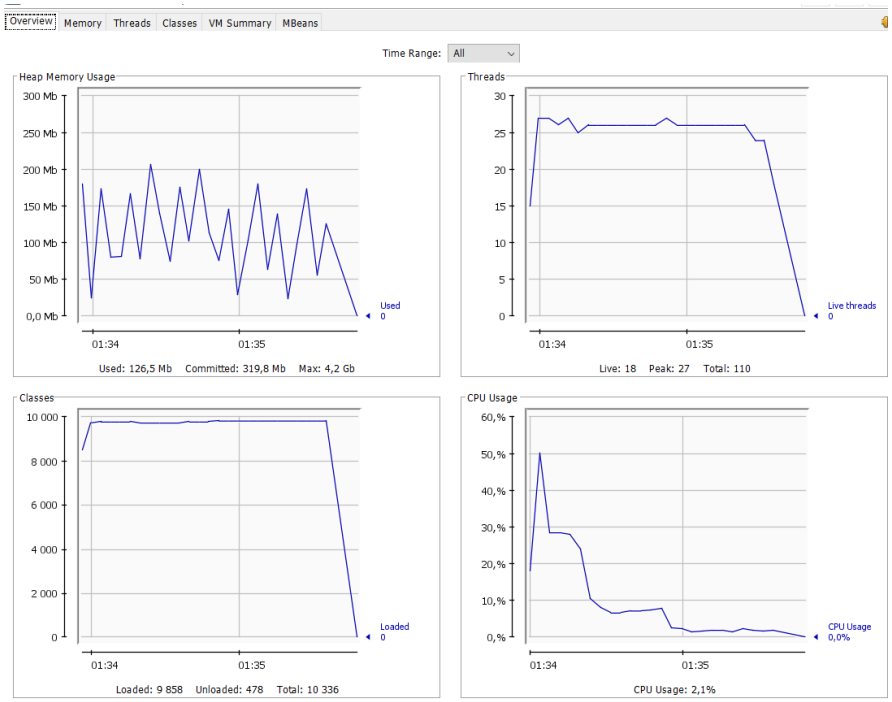

Рис. 3.15. Графічний вигляд статистики jconsole

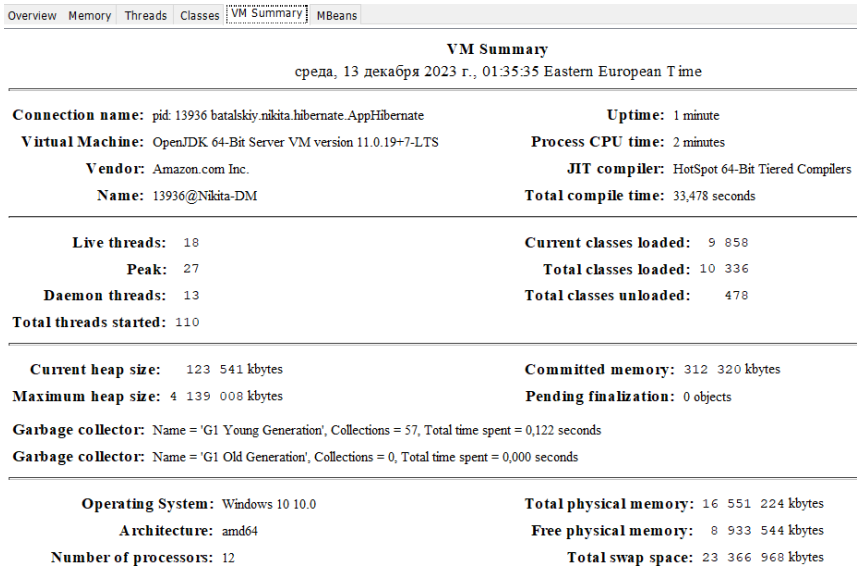

Рис. 3.16. Текстовий вигляд статистики jconsole

3.3. Результат третього запуску програми

Це вже третій запуск програми, тут буде менше коментарів. Час (рис. 3.17.) майже не відрізняється.

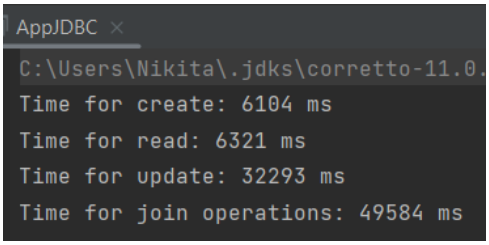

Рис. 3.17. Замір часу на виконання методів за допомогою jdbc

Графік БД у PG (рис. 3.18) майже такий самий, різниця тільки у кількості кортежів, на запис та оновлення їх стало трохи більше, а на зчитування – навпаки.

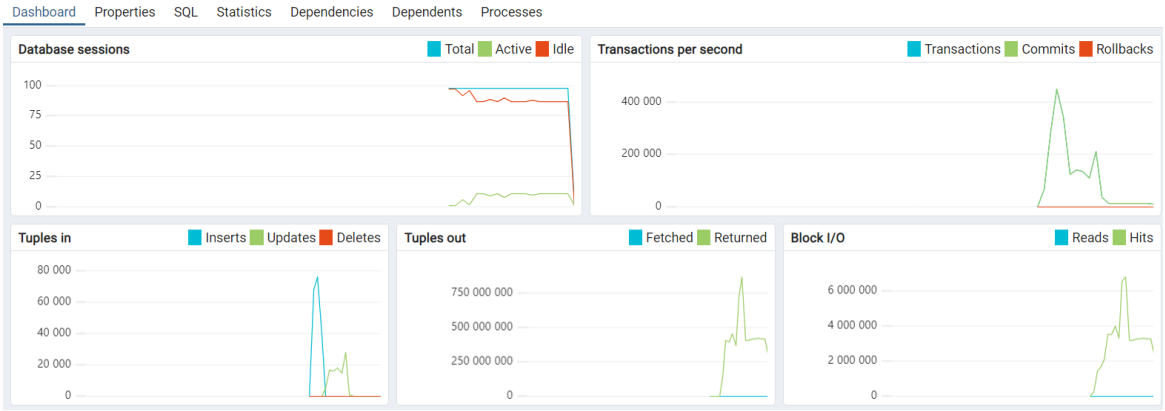

Рис. 3.18. Графік БД за час виконання методів

У jconsole (рис. 3.19, рис. 3.20.) ми бачимо вже знайомі графіки, які майже не відрізняються.

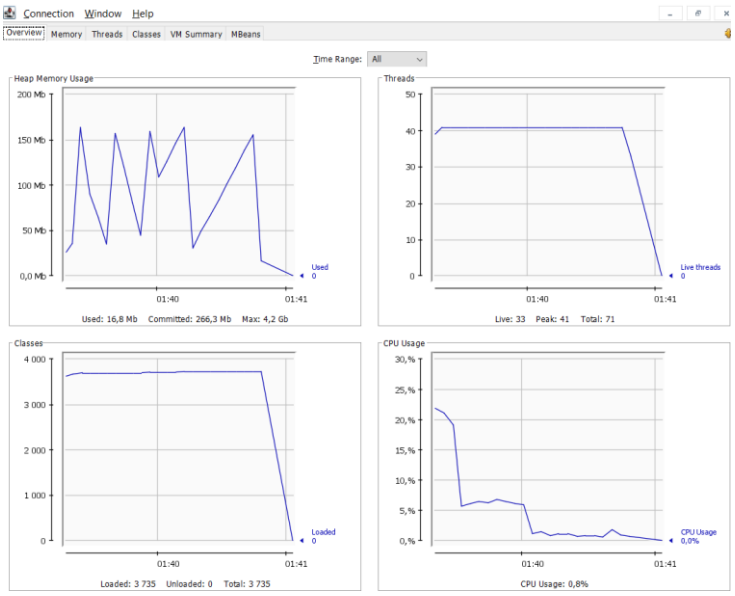

Рис. 3.19. Графічний вигляд статистики jconsole

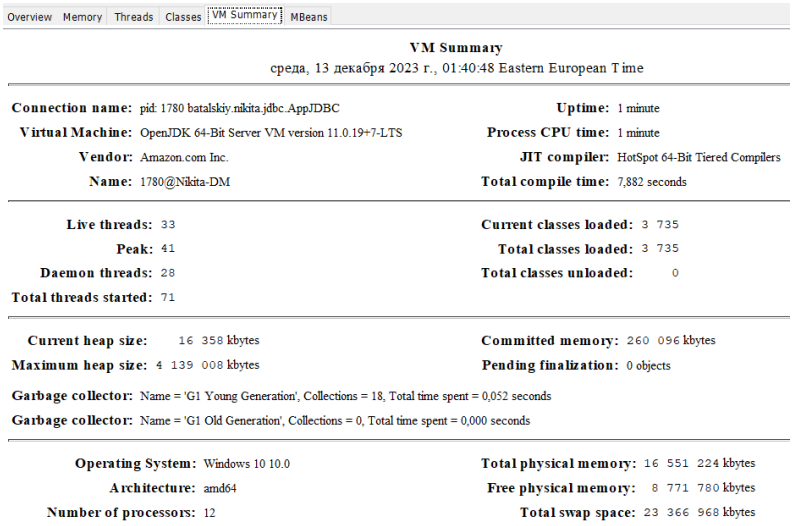

Рис. 3.20. Текстовий вигляд статистики jconsole

Тепер черга hibernate. Він знову показав себе загалом краще (рис. 3.21.).

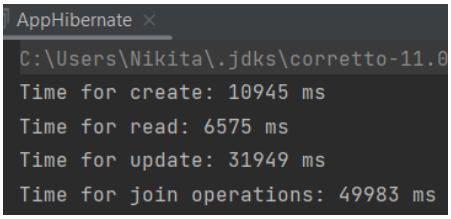

Рис. 3.21. Замір часу на виконання методів за допомогою hibernate

PG (рис. 3.22.) показує нам майже такий самий графік, тільки кортежів запису цього разу трохи більше, а зчитування – менше.

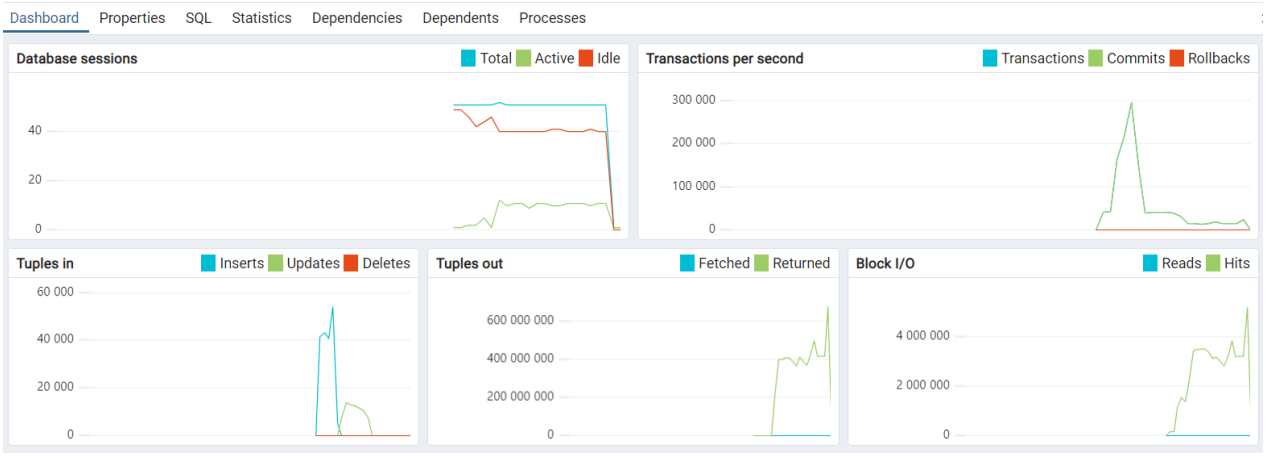

Рис. 3.22. Графік БД за час виконання методів

В свою чергу, jconsole (рис. 3.23., рис. 3.24.), показує ідентичний графік до другого запуску програми.

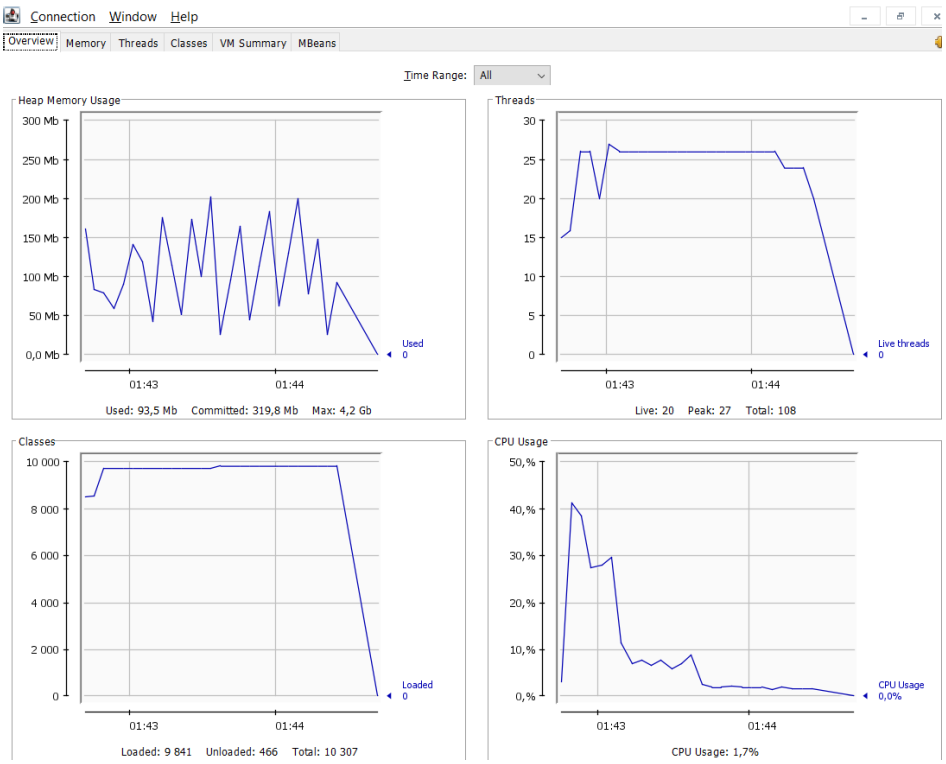

Рис. 3.23. Графічний вигляд статистики jconsole

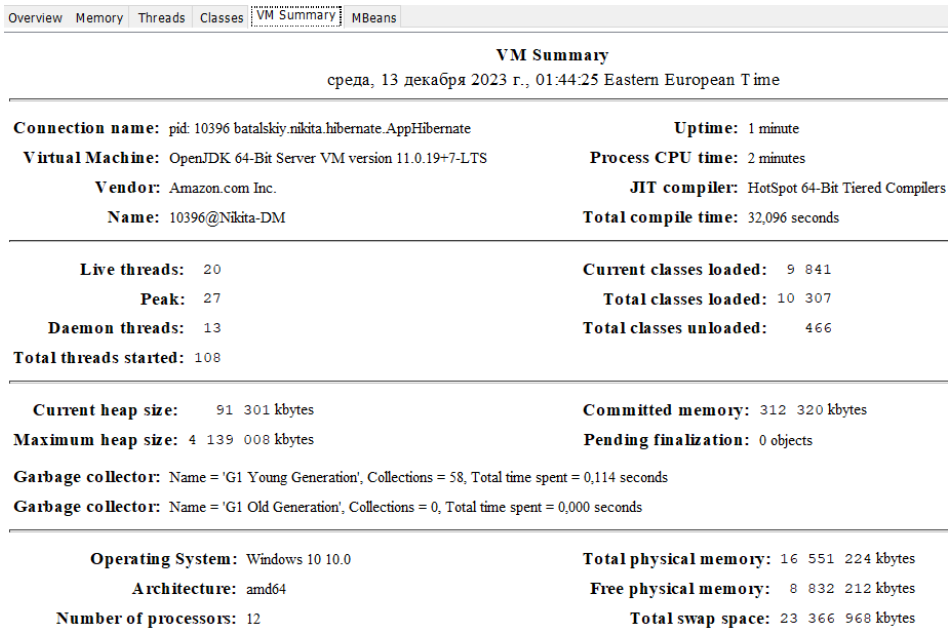

Рис. 3.24. Текстовий вигляд статистики jconsole

3.4. Підсумки за результатами запусків програми

Безумовно, існує похибка, яку можна побачити на рисунках вище, якщо порівнювати запуски реалізацій між собою, це обумовлено тим, що розроблені методи були вимушені обирати випадкові сутності з БД і могли натикатись кілька разів на одну й ту ж сутність.

Тож, якщо підводити підсумки, ми можемо сказати наступне: по часу на виконання – приблизно однакові результати; по статистиці БД – hibernate використав менше ресурсів; по jconsole – jdbc сторив більше потоків, проте в той же час створив менше класів (у hibernate це обумовлюється більш складною структурою). Варто також враховувати, що порівнюються технології з мінімумом оптимізацій, а також самі структурі сутностей доволі прості, а hibernate як раз отримує перевагу швидкості роботи у більш складних структурах даних.

ВИСНОВКИ

Метою кваліфікаційної роботи був аналіз та порівняння технологій jdbc та hibernate. Робота є актуальною за технологіями, фреймворками та бібліотеками.

Призначення розробленого застосунку полягає у полегшенні вибору тої чи іншої технології для взаємодії з БД розробниками, залежно від їх проекту.

Для розробки додатку використовувались сучасні підходи для створення програм з можливістю подальшого впровадження до інших систем. За необхідності застосунок може бути розширено новим функціоналом, якщо такий буде потрібен для покращення роботи додатку.

Порівняння JDBC і Hibernate може проводитися за різними критеріями, з урахуванням їх особливостей і призначення.

Рівень абстракції: JDBC – низький рівень абстракції, забезпечує прямий доступ до бази даних за допомогою SQL-запитів; Hibernate – високий рівень абстракції, приховує деталі взаємодії з базою даних, надаючи шар ORM (Object-Relational Mapping).

Рівень роботи з об'єктом: JDBC працює з реляційними даними за допомогою SQL-запитів, вимагає ручного перетворення даних між об'єктами бази даних і таблицями; Hibernate дозволяє працювати безпосередньо з об'єктами, автоматично зіставляючи об'єкти Java з відповідними таблицями в базі даних.

SQL-запити: У JDBC ви повинні явно писати SQL-запити для взаємодії з базою даних; Hibernate дозволяє використовувати HQL (Hibernate Query Language), який абстрагується від певної бази даних, а також надає можливість використовувати критерії та SQL-подібні запити.

Продуктивність: JDBC може бути більш ефективним у певних випадках, оскільки не вводить додаткових рівнів абстракції; Hibernate може спричинити деякі накладні витрати через перетворення ORM та додаткові запити, але забезпечує високий рівень зручності.

Складність розробки: Застосування JDBC вимагає більше зусиль для написання та підтримки SQL-запитів, а також перетворення даних між об'єктами та реляційною структурою; Hibernate забезпечує рівень абстракції, який спрощує розробку, особливо під час роботи з об'єктами.

Гнучкість і контроль: JDBC забезпечує більш прямий і повний контроль над запитами SQL і схемою бази даних; Hibernate забезпечує високий рівень абстракції, що може зменшити контроль, але також зменшити потребу в ручному кодуванні.

Підтримка посилань і каскадів: JDBC потребує ручного керування зв'язками та каскадуванням між об'єктами бази даних і таблицями; Hibernate забезпечує автоматичну підтримку посилань, включно з каскадуванням операцій.

Підтримка кешу: JDBC вимагає ручного налаштування та використання кешу при необхідності; Hibernate забезпечує вбудовану підтримку вторинного рівня кешу для покращення продуктивності.

Переносимість коду: JDBC більш портативний, якщо вам потрібно змінити базу даних; Hibernate надає абстракцію з бази даних, але може вимагати змін, якщо ви перейдете до іншої СУБД.

Вибір між JDBC і Hibernate залежить від вимог проекту, рівня контролю, гнучкості, рівня абстракції, досвіду команди розробників. У деяких випадках, особливо для простих додатків, може бути перевага JDBC, тоді як для складних і об'єктно-орієнтованих додатків Hibernate може надати більші переваги.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. E. Brewer A certain freedom: thoughts on the CAP theorem / E. Brewer. - in Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, New York, NY, USA. - 2010. - с. 335.

2. S. Nishida і Y. Shinkawa A Comprehensive Evaluation Model for BASE Transaction Processing / S. Nishida і Y. Shinkawa. - presented at the 9th International Conference on Software Engineering and Applications. - 2022. - с. 393–400.

3. S. Kul і A. Sayar A Survey of Publish/Subscribe Middleware Systems for Microservice Communication / S. Kul і A. Sayar. - in 2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). - 2021. - с. 781–785.

4. A. Diepenbrock, F. Rademacher, і S. Sachweh An Ontology-based Approach for Domain-driven Design of Microservice Architectures / . Gesellschaft für Informatik, Bonn, 2017.

5. E. Levy, H. F. Korth, і A. Silberschatz An optimistic commit protocol for distributed transaction management / E. Levy, H. F. Korth, і A. Silberschatz. - in Proceedings of the 1991 ACM SIGMOD international conference on Management of data, New York, NY, USA. - 1991. - с. 88–97.

6. H. Chandra Analysis of Change Data Capture Method in Heterogeneous Data Sources to Support RTDW / H. Chandra. - in 2018 4th International Conference on Computer and Information Sciences (ICCOINS). - 2018. - с. 1–6.

7. A. Karakos, D. Patsas, A. Bornea, і S. Kontogiannis Balancing HTTP Traffic Using Dynamically Updated Weights, an Implementation Approach / A. Karakos, D. Patsas, A. Bornea, і S. Kontogiannis. - in Advances in Informatics, Berlin, Heidelberg. - 2005. - с. 858–868.

8. D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, і N. C. Mendonça Beethoven: An Event-Driven Lightweight Platform for Microservice Orchestration / D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, і N. C. Mendonça. - in Software Architecture, Cham. - 2018. - с. 191–199.

9. T. Panpaliya Benchmarking MongoDB multi-document transactions in a sharded cluster / , Master of Science, San Jose State University, San Jose, CA, USA, 2020.

10. Maven [Електронний ресурс]. URL: https://maven.apache.org/guides/index.html

11. PostgreSQL [Електронний ресурс]. Режим доступу: https://www.postgresql.org/docs/

12. Lombok [Електронний ресурс]. URL: https://www.baeldung.com/intro-to-project-lombok

13. JConsole [Електронний ресурс]. Режим доступу: https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

14. HikariCP [Електронний ресурс]. URL: https://github.com/brettwooldridge/HikariCP

15. Logback [Електронний ресурс]. URL: https://www.baeldung.com/logback

16. PostgreSQL pg_dump [Електронний ресурс]. URL: https://www.postgresql.org/docs/current/app-pgdump.html

17. SQL JOIN [Електронний ресурс]. URL: https://w3schoolsua.github.io/sql/sql_join.html#gsc.tab=0

18. HikariCP [Електронний ресурс]. URL: https://www.baeldung.com/hikaricp

19. Basic entity annotations [Електронний ресурс]. URL: https://zetcode.com/springboot/annotations/#:~:text=The%20%40Entity%20annotation%20specifies%20that,to%20be%20used%20for%20mapping

20. Y. Chen і Z. He Bounds on the reliability of distributed systems with unreliable nodes & links / Y. Chen і Z. He. - IEEE Transactions on Reliability. - вип. 53, вип. 2, Чер 2004. - с. 205–215.

21. S. Gilbert і N. Lynch Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services / S. Gilbert і N. Lynch. - SIGACT News. - вип. 33, вип. 2, Чер 2002. - с. 51–59.

22. M. A. Fardbastani, F. Allahdadi, i M. Sharifi Business process monitoring via decentralized complex event processing / M. A. Fardbastani, F. Allahdadi, i M. Sharifi. - Enterprise Information Systems. - вип. 12, вип. 10, Лис 2018. - с. 1257–1284.

23. N. Naik Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP / N. Naik. - in 2017 IEEE International Systems Engineering Symposium (ISSE). - 2017. - с. 1–7.

24. C. K. Rudrabhatla Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture / C. K. Rudrabhatla. - International Journal of Advanced Computer Science and Applications (ijacsa). - вип. 9, вип. 8, 49/01 2018.

25. Anthony Molinaro. SQL Cookbook: Query Solutions and Techniques for Database Developers, 1st Edition. - O'Reilly Media, 2005. - 877 p.

26. Alan Beaulieu. Learning SQL: Master SQL Fundamentals, 2nd Edition. - O'Reilly Media, 2009. - 388p.

27. Itzik Ben-Gan. T-SQL Fundamentals, 3rd Edition. Microsoft Press, 2016. - 464p.

28. Stephane Faroult, Peter Robson. The art of SQL, 1st Edition. - O'Reilly Media, 2006. - 370 p.

29. Upon Malik, Matt Goldwasser, Benjamin Johnston. SQL for Data Analytics: Perform fast and efficient data analysis with the power of SQL, 1st Edition. - Packt Publishing, 2019. - 437p.

30. Upon Malik, Matt Goldwasser, Benjamin Johnston. The Applied SQL Data Analytics Workshop: Develop your practical skills and prepare to become a professional data analyst, 2nd Edition. - Packt Publishing, 2020. - 546p.

31. Michael Walker. Python Data Cleaning Cookbook: Modern techniques and Python tools to detect and remove dirty data and extract key insights, 1st Edition. - Packt Publishing, 2020. - 436p.

32. Felix Zumstein. Python for Excel: A Modern Environment for Automation and Data Analysis 1st Edition. - O'Reilly Media, 2021. - 565p.

33. Imran Ahmad. 40 Algorithms Every Programmer Should Know: Hone your problem-solving skills by learning different algorithms and their implementation in Python, 1st Edition. - Packt Publishing, 2020. - 384p.

34. Hong Zhou. Learn Data Mining Through Excel: A Step-by-Step Approach for Understanding Machine Learning Methods, 1st Edition. Apress, 2020. - 236p.

35. Stephen Klosterman. Data Science Projects with Python: A case study approach to successful data science projects using Python, pandas, and scikit-learn, 1st Edition. - Packt Publishing, 2019. - 374p.

.

ДОДАТОК А

КОД ПРОГРАМИ

Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>batalskiy.nikita</groupId>
  <artifactId>graduateWork</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>graduateWork</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <!-- PostgreSQL -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.5.1</version>
    </dependency>
    <!-- Hibernate -->
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.3.1.Final</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.6.14.Final</version>
    </dependency>
    <!-- Lombok -->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.30</version>
      <scope>provided</scope>
```

```
    </dependency>
    <!--Plugin -->
    <dependency>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
    </dependency>
    <!-- Logging -->
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.4.5</version>
    </dependency>
    <!-- HikariCP -->
    <dependency>
      <groupId>com.zaxxer</groupId>
      <artifactId>HikariCP</artifactId>
      <version>5.0.1</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-hikaricp</artifactId>
      <version>6.3.1.Final</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.hikari.dataSourceClassName">org.postgresql.ds.PGSimpleDataSource</property>
    <property name="hibernate.hikari.dataSource.url">jdbc:postgresql://localhost/hibernate</property>
    <property name="hibernate.hikari.dataSource.user">postgres</property>
    <property name="hibernate.hikari.dataSource.password">653241</property>
    <property name="current_session_context_class">thread</property>
    <property name="show_sql">false</property>
    <property name="format_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <property name="hibernate.hikari.maximumPoolSize">50</property>
    <mapping class="batalskiy.nikita.hibernate.model.User" />
    <mapping class="batalskiy.nikita.hibernate.model.Order" />
    <mapping class="batalskiy.nikita.hibernate.model.OrderItem" />
    <mapping class="batalskiy.nikita.hibernate.model.Product" />
  </session-factory>
</hibernate-configuration>
```

Logback.xml

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>
  <logger name="org.hibernate" level="WARN" additivity="false">
    <appender-ref ref="STDOUT"/>
  </logger>
  <logger name="com.zaxxer.hikari" level="WARN" additivity="false">
    <appender-ref ref="STDOUT"/>
  </logger>
  <root level="ERROR">
    <appender-ref ref="STDOUT"/>
  </root>
  <logger name="com.zaxxer.hikari" level="WARN"/>
</configuration>
```

DatabaseConfig.java

```
public class DatabaseConfig {
  public static final String URL = "jdbc:postgresql://localhost/jdbc";
  public static final String USERNAME = "postgres";
  public static final String PASSWORD = "653241";
```

```
  private static HikariDataSource dataSource;

  public static DataSource getDataSource() {
    HikariConfig hikariConfig = new HikariConfig();
    hikariConfig.setJdbcUrl(URL);
    hikariConfig.setUsername(USERNAME);
    hikariConfig.setPassword(PASSWORD);
    hikariConfig.setMaximumPoolSize(50);
    dataSource = new HikariDataSource(hikariConfig);
    return dataSource;
  }
}
```

OrderDao.java

```
public class OrderDao implements OrderRepository {
  private final DataSource dataSource;
  private final UserDao userDao;

  public OrderDao() {
    this.dataSource = DatabaseConfig.getDataSource();
    this.userDao = new UserDao();
  }

  @Override
  public Order getOrderById(long orderId) {
    Order order = null;
    try (Connection connection = dataSource.getConnection()) {
      PreparedStatement stmt =
          connection.prepareStatement("SELECT * FROM jdbc.public.order WHERE id = ?");
      stmt.setLong(1, orderId);
      ResultSet rs = stmt.executeQuery();
      if (rs.next()) {
        order = mapOrder(rs);
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return order;
  }

  @Override
  public List<Order> getOrdersByUserId(long userId) {
    ArrayList<Order> orders = new ArrayList<>();
    try (Connection connection = dataSource.getConnection()) {
      PreparedStatement stmt =
          connection.prepareStatement("SELECT * FROM jdbc.public.order WHERE user_id = ?");
      stmt.setLong(1, userId);
      ResultSet rs = stmt.executeQuery();

      while (rs.next()) {
        orders.add(mapOrder(rs));
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return orders;
  }

  @Override
  public ArrayList<Order> getAllOrders() {
    ArrayList<Order> orders = new ArrayList<>();
```

```
  try (Connection connection = dataSource.getConnection()) {
    PreparedStatement stmt = connection.prepareStatement("SELECT * FROM jdbc.public.order");
    ResultSet rs = stmt.executeQuery();

    while (rs.next()) {
      orders.add(mapOrder(rs));
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
  return orders;
}

@Override
public void addOrder(Order order) {
  try (Connection connection = dataSource.getConnection()) {
    PreparedStatement stmt =
        connection.prepareStatement(
            "INSERT INTO jdbc.public.order (user_id, date, status) VALUES (?,?,?)");
    stmt.setLong(1, order.getUser().getId());
    stmt.setObject(2, order.getDate());
    stmt.setString(3, order.getStatus());
    stmt.executeUpdate();
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

@Override
public void updateOrder(Order order) {
  try (Connection connection = dataSource.getConnection()) {
    if (getOrderById(order.getId()) != null) {
      PreparedStatement stmt =
          connection.prepareStatement("UPDATE jdbc.public.order SET status = ? WHERE id = ?");
      stmt.setString(1, order.getStatus());
      stmt.setLong(2, order.getId());

      stmt.executeUpdate();
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

@Override
public void deleteOrder(long orderId) {
  try (Connection connection = dataSource.getConnection()) {
    if (getOrderById(orderId) != null) {
      PreparedStatement stmt =
          connection.prepareStatement("DELETE FROM jdbc.public.order WHERE id = ?");

      stmt.setLong(1, orderId);
      stmt.executeUpdate();
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

public long findMaxId() {
  try (Connection connection = dataSource.getConnection()) {
    PreparedStatement stmt =
```

```
        connection.prepareStatement("SELECT id FROM jdbc.public.order ORDER BY id DESC LIMIT 1");

      ResultSet resultSet = stmt.executeQuery();
      if (resultSet.next()) {
        return resultSet.getLong("id");
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }

    return 0;
  }

  public ArrayList<String> getOrdersWithDetails(long userId) {
    ArrayList<String> orders = new ArrayList<>();
    long id = 0;
    long user_id = 0;
    LocalDate date = null;
    String status = null;
    String productName = null;
    int price = 0;
    try (Connection connection = dataSource.getConnection()) {
      PreparedStatement stmt =
          connection.prepareStatement(
              "SELECT o.id, o.user_id, o.date, o.status, p.name, p.price "
                  + "FROM jdbc.public.order o "
                  + "JOIN jdbc.public.order_item i ON o.id = i.order_id "
                  + "JOIN jdbc.public.product p ON i.product_id = p.id "
                  + "WHERE o.user_id = ?");
      stmt.setLong(1, userId);
      ResultSet rs = stmt.executeQuery();
      while (rs.next()) {
        id = rs.getLong("id");
        user_id = rs.getLong("user_id");
        date = rs.getObject("date", LocalDate.class);
        status = rs.getString("status");
        productName = rs.getString("name");
        price = rs.getInt("price");
        StringBuilder result = new StringBuilder();
        result.append("order id = ").append(id)
            .append("\norder user_id = ").append(user_id)
            .append("\norder date = ").append(date)
            .append("\norder status = ").append(status)
            .append("\nproduct name = ").append(productName)
            .append("\nproduct price = ").append(price);
        orders.add(result.toString());
      }
    } catch (SQLException e) {
      System.out.println(e.getMessage());
    }
    return orders;
  }

  private Order mapOrder(ResultSet rs) throws SQLException {
    Order order = new Order();
    order.setId(rs.getLong("id"));
    order.setUser(userDao.getUserById(rs.getLong("user_id")));
    order.setDate(rs.getObject("date", LocalDate.class));
    order.setStatus(rs.getString("status"));
    return order;
  }
}
```

```
public class OrderItemDao implements OrderItemRepository {
  private final DataSource dataSource;
  private final OrderDao orderDao;
  private final ProductDao productDao;

  public OrderItemDao( ) {
    this.dataSource = DatabaseConfig.getDataSource();
    this.orderDao = new OrderDao();
    this.productDao = new ProductDao();
  }

  @Override
  public OrderItem getOrderItemById(long orderItemId) {
    OrderItem orderItem = null;
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement("SELECT * FROM jdbc.public.order_item WHERE id = ?");
      stmt.setLong(1, orderItemId);

      ResultSet rs = stmt.executeQuery();

      if (rs.next()) {
        orderItem = mapOrderItem(rs);
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return orderItem;
  }

  @Override
  public List<OrderItem> getOrderItemsByOrderId(long orderId) {
    ArrayList<OrderItem> orderItems = new ArrayList<>();
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement("SELECT * FROM jdbc.public.order_item WHERE order_id = ?");
      stmt.setLong(1, orderId);
      ResultSet rs = stmt.executeQuery();

      while (rs.next()) {
        orderItems.add(mapOrderItem(rs));
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return orderItems;
  }

  @Override
  public void addOrderItem(OrderItem orderItem) {
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement(
              "INSERT INTO jdbc.public.order_item (order_id, product_id, quantity) VALUES (?,?,?)");
      stmt.setLong(1, orderItem.getOrder().getId());
      stmt.setLong(2, orderItem.getProduct().getId());
      stmt.setInt(3, orderItem.getQuantity());
      stmt.executeUpdate();
    } catch (SQLException e) {
```

```
      throw new RuntimeException(e);
    }
  }

  @Override
  public void updateOrderItem(OrderItem orderItem) {
    try (Connection connection = dataSource.getConnection()){
      if (getOrderItemById(orderItem.getId()) != null) {
        PreparedStatement stmt =
            connection.prepareStatement(
                "UPDATE jdbc.public.order_item SET quantity = ? WHERE id = ?");
        stmt.setInt(1, orderItem.getQuantity());
        stmt.setLong(2, orderItem.getId());

        stmt.executeUpdate();
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  public void deleteOrderItem(long orderItemId) {
    try (Connection connection = dataSource.getConnection()){
      if (getOrderItemById(orderItemId) != null) {
        PreparedStatement stmt =
            connection.prepareStatement("DELETE FROM jdbc.public.order_item WHERE id = ?");

        stmt.setLong(1, orderItemId);
        stmt.executeUpdate();
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  public long findMaxId(){
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement("SELECT id FROM jdbc.public.order_item ORDER BY id DESC LIMIT
1");

      ResultSet resultSet = stmt.executeQuery();
      if (resultSet.next()) {
        return resultSet.getLong("id");
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }

    return 0;
  }

  private OrderItem mapOrderItem(ResultSet rs) throws SQLException {
    OrderItem orderItem = new OrderItem();
    orderItem.setId(rs.getLong("id"));
    orderItem.setOrder(orderDao.getOrderById(rs.getLong("order_id")));
    orderItem.setProduct(productDao.getProductById(rs.getLong("product_id")));
    orderItem.setQuantity(rs.getInt("quantity"));
    return orderItem;
  }
}
```

```
public class ProductDao implements ProductRepository {
  private final DataSource dataSource;

  public ProductDao() {
    this.dataSource = DatabaseConfig.getDataSource();
  }

  @Override
  public Product getProductById(long productId) {
    Product product = null;
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement("SELECT * FROM jdbc.public.product WHERE id = ?");
      stmt.setLong(1, productId);

      ResultSet rs = stmt.executeQuery();

      if (rs.next()) {
        product = mapProduct(rs);
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return product;
  }

  @Override
  public List<Product> getAllProducts() {
    ArrayList<Product> products = new ArrayList<>();
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt = connection.prepareStatement("SELECT * FROM jdbc.public.product");
      ResultSet rs = stmt.executeQuery();

      while (rs.next()) {
        products.add(mapProduct(rs));
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return products;
  }

  @Override
  public void addProduct(Product product) {
    try (Connection connection = dataSource.getConnection()){
      PreparedStatement stmt =
          connection.prepareStatement(
              "INSERT INTO jdbc.public.product (name, description, price) VALUES (?,?,?)");
      stmt.setString(1, product.getName());
      stmt.setString(2, product.getDescription());
      stmt.setDouble(3, product.getPrice());
      stmt.executeUpdate();
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
  }

  @Override
  public void updateProduct(Product product) {
    try (Connection connection = dataSource.getConnection()){
```

```
    if (getProductById(product.getId()) != null) {
      PreparedStatement stmt =
          connection.prepareStatement(
              "UPDATE jdbc.public.product SET name = ?, description = ?, price = ? WHERE id = ?");
      stmt.setString(1, product.getName());
      stmt.setString(2, product.getDescription());
      stmt.setDouble(3, product.getPrice());
      stmt.setLong(4, product.getId());

      stmt.executeUpdate();
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
 }

 @Override
 public void deleteProduct(long productId) {
  try (Connection connection = dataSource.getConnection()){
    if (getProductById(productId) != null) {
      PreparedStatement stmt =
          connection.prepareStatement("DELETE FROM jdbc.public.product WHERE id = ?");

      stmt.setLong(1, productId);
      stmt.executeUpdate();
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
 }

 public long findMaxId(){
  try (Connection connection = dataSource.getConnection()){
    PreparedStatement stmt =
        connection.prepareStatement("SELECT id FROM jdbc.public.product ORDER BY id DESC LIMIT 1");

    ResultSet resultSet = stmt.executeQuery();
    if (resultSet.next()) {
      return resultSet.getLong("id");
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }

  return 0;
 }

 private Product mapProduct (ResultSet rs) throws SQLException {
  Product product = new Product();
  product.setId(rs.getLong("id"));
  product.setName(rs.getString("name"));
  product.setDescription(rs.getString("description"));
  product.setPrice(rs.getInt("price"));
  return product;
 }
}
```

UserDao.java

```
public class UserDao implements UserRepository {

 private final DataSource dataSource;
```

```
public UserDao() {
  this.dataSource = DatabaseConfig.getDataSource();
}

public User getUserById(long userId) {
  User user = null;
  try (Connection connection = dataSource.getConnection()){
    PreparedStatement stmt =
        connection.prepareStatement("SELECT * FROM jdbc.public.user WHERE id = ?");
    stmt.setLong(1, userId);

    ResultSet rs = stmt.executeQuery();

    if (rs.next()) {
      user = mapUser(rs);
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
  return user;
}

public List<User> getAllUsers() {
  ArrayList<User> users = new ArrayList<>();
  try (Connection connection = dataSource.getConnection()){
    PreparedStatement stmt = connection.prepareStatement("SELECT * FROM jdbc.public.user");
    ResultSet rs = stmt.executeQuery();

    while (rs.next()) {
      users.add(mapUser(rs));
    }
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
  return users;
}

public void addUser(User user) {
  try (Connection connection = dataSource.getConnection()){
    PreparedStatement stmt =
        connection.prepareStatement(
            "INSERT INTO jdbc.public.user (name, lastname, address, email) VALUES (?,?,?,?)");
    stmt.setString(1, user.getName());
    stmt.setString(2, user.getLastName());
    stmt.setString(3, user.getAddress());
    stmt.setString(4, user.getEmail());
    stmt.executeUpdate();
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
}

public void updateUser(User user) {
  try (Connection connection = dataSource.getConnection()){
    if (getUserById(user.getId()) != null) {
      PreparedStatement stmt =
          connection.prepareStatement(
              "UPDATE jdbc.public.user SET name = ?, lastname = ?, address = ?,  email = ? WHERE id = ?");
      stmt.setString(1, user.getName());
      stmt.setString(2, user.getLastName());
      stmt.setString(3, user.getAddress());
      stmt.setString(4, user.getEmail());
```

```
        stmt.setLong(5, user.getId());

        stmt.executeUpdate();
       }
     } catch (SQLException e) {
       throw new RuntimeException(e);
     }
   }

   public void deleteUser(long userId) {
     try (Connection connection = dataSource.getConnection()){
       if (getUserById(userId) != null) {
         PreparedStatement stmt =
             connection.prepareStatement("DELETE FROM jdbc.public.user WHERE id = ?");

         stmt.setLong(1, userId);
         stmt.executeUpdate();
       }
     } catch (SQLException e) {
       throw new RuntimeException(e);
     }
   }

   public long findMaxId(){
     try (Connection connection = dataSource.getConnection()){
         PreparedStatement stmt =
             connection.prepareStatement("SELECT id FROM jdbc.public.user ORDER BY id DESC LIMIT 1");

       ResultSet resultSet = stmt.executeQuery();
       if (resultSet.next()) {
         return resultSet.getLong("id");
       }
     } catch (SQLException e) {
       throw new RuntimeException(e);
     }

     return 0;
   }

   private User mapUser (ResultSet rs) throws SQLException {
     User user = new User();
     user.setId(rs.getLong("id"));
     user.setName(rs.getString("name"));
     user.setLastName(rs.getString("lastname"));
     user.setAddress(rs.getString("address"));
     user.setEmail(rs.getString("email"));
     return user;
   }
}
```

AppJDBC.java

```
public class AppJDBC {
  static UserService userService = new UserService();
  static OrderService orderService = new OrderService();
  static OrderItemService orderItemService = new OrderItemService();
  static ProductService productService = new ProductService();

public static void main(String[] args) {
    for (int i = 1; i <= 50000; i++) {
      User user = createUser(i);
      userService.addUser(user);
    }
```

```
  for (int i = 1; i <= 50000; i++) {
    Product product = createProduct(i);
    productService.addProduct(product);
  }

  createOrdersAndOrderItemsForTest();
  testReadPerformance();
  testUpdatePerformance();
  testJoinPerformance();
}

private static void createOrdersAndOrderItemsForTest() {
  ExecutorService executor = Executors.newFixedThreadPool(10);

  long start = System.currentTimeMillis();

  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            Order order = generateTestOrder();
            orderService.addOrder(order);

            OrderItem item = generateTestOrderItem();
            orderItemService.addOrderItem(item);
            iterationCount++;
          }
        });
  }

  executor.shutdown();

  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }

  long time = System.currentTimeMillis() - start;
  System.out.println("Time for create: " + time + " ms");
}

private static void testReadPerformance() {

  ExecutorService executor = Executors.newFixedThreadPool(10);

  long start = System.currentTimeMillis();

  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            getRandomOrder();
            getRandomOrderItem();
            getRandomUser();
            getRandomProduct();
            iterationCount++;
          }
        });
```

```
  }

  executor.shutdown();
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for read: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static void testJoinPerformance() {

  ExecutorService executor = Executors.newFixedThreadPool(10);

  long start = System.currentTimeMillis();

  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            orderService.getOrdersWithDetails(getRandomUser().getId());
            iterationCount++;
          }
        });
  }

  executor.shutdown();
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for join operations: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static void testUpdatePerformance() {
  ExecutorService executor = Executors.newFixedThreadPool(10);
  long start = System.currentTimeMillis();
  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            Order order = getRandomOrder();
            order.setStatus("In work");
            orderService.updateOrder(order);
            ArrayList<OrderItem> orderItemsByOrderId =
                (ArrayList<OrderItem>) orderItemService.getOrderItemsByOrderId(order.getId());
            orderItemsByOrderId.forEach(
                orderItem -> {
                  orderItem.setQuantity((int) (Math.random() * 9) + 1);
                  orderItemService.updateOrderItem(orderItem);
                });
            iterationCount++;
          }
        });
  }
  executor.shutdown();
```

```
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for update: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static Product createProduct(int i) {
  Product product = new Product();
  product.setName("Product " + i);
  product.setDescription("Product description " + i);
  product.setPrice(i);
  return product;
}

private static User createUser(int i) {
  User user = new User();
  user.setName("Name " + i);
  user.setLastName("LastName " + i);
  user.setAddress("Address " + i);
  user.setEmail("Email " + i);
  return user;
}

private static User getRandomUser() {
  User user = null;
  while (user == null) {
    long userId = (long) ((Math.random() * (userService.findMaxId() - 1)) + 1);
    user = userService.getUserById(userId);
  }
  return user;
}

private static Product getRandomProduct() {
  Product product = null;
  while (product == null) {
    long productId = (long) ((Math.random() * (productService.findMaxId() - 1)) + 1);
    product = productService.getProductById(productId);
  }
  return product;
}

private static Order getRandomOrder() {
  Order order = null;
  while (order == null) {
    long orderId = (long) ((Math.random() * (orderService.findMaxId() - 1)) + 1);
    order = orderService.getOrderById(orderId);
  }
  return order;
}

private static OrderItem getRandomOrderItem() {
  OrderItem orderItem = null;
  while (orderItem == null) {
    long orderItemId = (long) ((Math.random() * (orderItemService.findMaxId() - 1)) + 1);
    orderItem = orderItemService.getOrderItemById(orderItemId);
  }
  return orderItem;
}
```

```
    private static Order generateTestOrder() {
      Order order = new Order();
      order.setUser(getRandomUser());
      order.setDate(LocalDate.now());
      order.setStatus("New");
      return order;
    }

    private static OrderItem generateTestOrderItem() {
      OrderItem orderItem = new OrderItem();
      orderItem.setOrder(getRandomOrder());
      orderItem.setProduct(getRandomProduct());
      orderItem.setQuantity((int) (Math.random() * 9) + 1);
      return orderItem;
    }
  }
```

HibernateUtil.java

```
public class HibernateUtil {

  private static SessionFactory sessionFactory = buildSessionFactory();


  protected static SessionFactory buildSessionFactory() {

    final StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

    try {

      sessionFactory = new MetadataSources(registry).buildMetadata().buildSessionFactory();

    } catch (Exception e) {

      StandardServiceRegistryBuilder.destroy(registry);

      throw new ExceptionInInitializerError("Initial SessionFactory failed" + e);

    }

    return sessionFactory;

  }


  public static SessionFactory getSessionFactory() {

    return sessionFactory;

  }

}
```

OrderDao.java

```
public class OrderDao implements OrderRepository {
  private final SessionFactory sessionFactory;

  public OrderDao() {
    this.sessionFactory = HibernateUtil.getSessionFactory();
  }

  @Override
```

```
public Order getOrderById(long orderId) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    Order order = session.get(Order.class, orderId);
    session.getTransaction().commit();
    return order;
  }
}

@Override
public List<Order> getOrdersByUserId(long userId) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    ArrayList<Order> orders =
        (ArrayList<Order>)
            session
                .createQuery("FROM Order o WHERE o.user.id = :user_id", Order.class)
                .setParameter("user_id", userId)
                .getResultList();
    session.getTransaction().commit();
    return orders;
  }
}

@Override
public List<Order> getAllOrders() {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    ArrayList<Order> orders =
        (ArrayList<Order>) session.createQuery("FROM Order", Order.class).getResultList();
    session.getTransaction().commit();
    return orders;
  }
}

@Override
public void addOrder(Order order) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    session.merge(order);
    session.getTransaction().commit();
  }
}

@Override
public void updateOrder(Order order) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    Order orderToUpdate = session.get(Order.class, order.getId());
    orderToUpdate.setUser(order.getUser());
    orderToUpdate.setStatus(order.getStatus());
    session.merge(orderToUpdate);
    session.getTransaction().commit();
  }
}

@Override
public void deleteOrder(long orderId) {
```

```
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Order orderToDelete = session.get(Order.class, orderId);
      session.remove(orderToDelete);
      session.getTransaction().commit();
    }
  }

  public List<String> getOrdersWithDetails(long userId) {
    List<String> orders = new ArrayList<>();
    try (Session session = sessionFactory.openSession()) {

      List<Object[]> resultList = session.createQuery(
          "SELECT o.id, o.user.id, o.date, o.status, p.name, p.price "
              + "FROM Order o "
              + "JOIN OrderItem i on i.order.id = o.id "
              + "JOIN i.product p "
              + "WHERE o.user.id = :userId", Object[].class).setParameter("userId", userId).getResultList();

      for (Object[] row : resultList) {
        long id = (Long) row[0];
        long user_id = (Long) row[1];
        LocalDate date = (LocalDate) row[2];
        String status = (String) row[3];
        String productName = (String) row[4];
        int price = (Integer) row[5];

        StringBuilder resultString = new StringBuilder();
        resultString.append("order id = ").append(id)
            .append("\norder user_id = ").append(user_id)
            .append("\norder date = ").append(date)
            .append("\norder status = ").append(status)
            .append("\nproduct name = ").append(productName)
            .append("\nproduct price = ").append(price);
        orders.add(resultString.toString());
      }
    } catch (HibernateException e) {
      System.out.println(e.getMessage());
    }
    return orders;
  }

  public long getMaxOrderId() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Query<Long> query = session.createQuery("SELECT MAX(o.id) FROM Order o", Long.class);
      Long maxId = query.uniqueResult();
      session.getTransaction().commit();
      return maxId != null ? maxId : 0L;
    }
  }
}
```

OrderItemDao.java

```
public class OrderItemDao implements OrderItemRepository {
  private final SessionFactory sessionFactory;
```

```
public OrderItemDao() {
  this.sessionFactory = HibernateUtil.getSessionFactory();
}

@Override
public OrderItem getOrderItemById(long orderItemId) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    OrderItem orderItem = session.get(OrderItem.class, orderItemId);
    session.getTransaction().commit();
    return orderItem;
  }
}

@Override
public List<OrderItem> getAllOrderItems() {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    ArrayList<OrderItem> orderItems =
        (ArrayList<OrderItem>)
            session.createQuery("FROM OrderItem", OrderItem.class).getResultList();
    session.getTransaction().commit();
    return orderItems;
  }
}

@Override
public List<OrderItem> getOrderItemsByOrderId(long orderId) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    ArrayList<OrderItem> orderItems =
        (ArrayList<OrderItem>)
            session
                .createQuery("FROM OrderItem WHERE order.id = :order_id", OrderItem.class)
                .setParameter("order_id", orderId)
                .getResultList();
    session.getTransaction().commit();
    return orderItems;
  }
}

@Override
public void addOrderItem(OrderItem orderItem) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    session.merge(orderItem);
    session.getTransaction().commit();
  }
}

@Override
public void updateOrderItem(OrderItem orderItem) {
  try (Session session = sessionFactory.openSession()) {
    session.beginTransaction();
    OrderItem orderItemToUpdate = session.get(OrderItem.class, orderItem.getId());
    orderItemToUpdate.setProduct(orderItem.getProduct());
    orderItemToUpdate.setQuantity(orderItem.getQuantity());
    session.merge(orderItemToUpdate);
```

```
      session.getTransaction().commit();
    }
  }

  @Override
  public void deleteOrderItem(long orderItemId) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      OrderItem orderItemToDelete = session.get(OrderItem.class, orderItemId);
      session.remove(orderItemToDelete);
      session.getTransaction().commit();
    }
  }

  public long getMaxOrderItemId() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Query<Long> query = session.createQuery("SELECT MAX(id) FROM OrderItem", Long.class);
      Long maxId = query.uniqueResult();
      session.getTransaction().commit();
      return maxId != null ? maxId : 0L;
    }
  }
}
```

ProductDao.java

```
public class ProductDao implements ProductRepository {
  private final SessionFactory sessionFactory;

  public ProductDao() {
    this.sessionFactory = HibernateUtil.getSessionFactory();
  }

  @Override
  public Product getProductById(long productId) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Product product = session.get(Product.class, productId);
      session.getTransaction().commit();
      return product;
    }
  }

  @Override
  public List<Product> getAllProducts() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      ArrayList<Product> products =
          (ArrayList<Product>) session.createQuery("FROM Product", Product.class).getResultList();
      session.getTransaction().commit();
      return products;
    }
  }

  @Override
  public void addProduct(Product product) {
    try (Session session = sessionFactory.openSession()) {
```

```
      session.beginTransaction();
      session.merge(product);
      session.getTransaction().commit();
    }
  }

  @Override
  public void updateProduct(Product product) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Product productToUpdate = session.get(Product.class, product.getId());
      productToUpdate.setName(product.getName());
      productToUpdate.setDescription(product.getDescription());
      productToUpdate.setPrice(product.getPrice());
      session.merge(productToUpdate);
      session.getTransaction().commit();
    }
  }

  @Override
  public void deleteProduct(long productId) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Product productToDelete = session.get(Product.class, productId);
      session.remove(productToDelete);
      session.getTransaction().commit();
    }
  }

  public long getMaxProductId() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Query<Long> query = session.createQuery("SELECT MAX(id) FROM Product", Long.class);
      Long maxId = query.uniqueResult();
      session.getTransaction().commit();
      return maxId != null ? maxId : 0L;
    }
  }
}
```

UserDao.java

```
public class UserDao implements UserRepository {

  private final SessionFactory sessionFactory;

  public UserDao() {
    this.sessionFactory = HibernateUtil.getSessionFactory();
  }

  @Override
  public User getUserById(long userId) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      User user = session.get(User.class, userId);
      session.getTransaction().commit();
      return user;
    }
```

```
  }

  @Override
  public ArrayList<User> getAllUsers() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      ArrayList<User> users =
          (ArrayList<User>) session.createQuery("FROM User", User.class).getResultList();
      session.getTransaction().commit();
      return users;
    }
  }

  @Override
  public void addUser(User user) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      session.merge(user);
      session.getTransaction().commit();
    }
  }

  @Override
  public void updateUser(User user) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      User userToUpdate = session.get(User.class, user.getId());
      userToUpdate.setName(user.getName());
      userToUpdate.setLastName(user.getLastName());
      userToUpdate.setAddress(user.getAddress());
      userToUpdate.setEmail(user.getEmail());
      session.merge(userToUpdate);
      session.getTransaction().commit();
    }
  }

  @Override
  public void deleteUser(long userId) {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      User userToDelete = session.get(User.class, userId);
      session.remove(userToDelete);
      session.getTransaction().commit();
    }
  }

  public long getMaxUserId() {
    try (Session session = sessionFactory.openSession()) {
      session.beginTransaction();
      Query<Long> query = session.createQuery("SELECT MAX(id) FROM User ", Long.class);
      Long maxId = query.uniqueResult();
      session.getTransaction().commit();
      return maxId != null ? maxId : 0L;
    }
  }
}
```

```
public class AppHibernate {

  static UserService userService = new UserService();
  static ProductService productService = new ProductService();
  static OrderService orderService = new OrderService();
  static OrderItemService orderItemService = new OrderItemService();

public static void main(String[] args) {
    for (int i = 1; i <= 50000; i++) {
      User user = createUser(i);
      userService.addUser(user);
    }

    for (int i = 1; i <= 50000; i++) {
      Product product = createProduct(i);
      productService.addProduct(product);
    }

    createOrdersAndOrderItemsForTest();
    testReadPerformance();
    testUpdatePerformance();
    testJoinPerformance();
  }

  private static void createOrdersAndOrderItemsForTest() {
    ExecutorService executor = Executors.newFixedThreadPool(10);

    long start = System.currentTimeMillis();

    for (int i = 0; i < 10; i++) {
      executor.submit(
          () -> {
            int iterationCount = 0;
            while (iterationCount < 5000) {
              Order order = generateTestOrder();
              orderService.addOrder(order);

              OrderItem item = generateTestOrderItem();
              orderItemService.addOrderItem(item);
              iterationCount++;
            }
          });
    }
    executor.shutdown();
    try {
      executor.awaitTermination(5, TimeUnit.MINUTES);
      long time = System.currentTimeMillis() - start;
      System.out.println("Time for create: " + time + " ms");
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  }

  private static void testReadPerformance() {

    ExecutorService executor = Executors.newFixedThreadPool(10);

    long start = System.currentTimeMillis();
```

```
  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            getRandomOrder();
            getRandomOrderItem();
            getRandomUser();
            getRandomProduct();
            iterationCount++;
          }
        });
  }

  executor.shutdown();
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for read: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static void testJoinPerformance() {

  ExecutorService executor = Executors.newFixedThreadPool(10);
  long start = System.currentTimeMillis();

  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
          while (iterationCount < 5000) {
            orderService.getOrdersWithDetails(getRandomUser().getId());
            iterationCount++;
          }
        });
  }

  executor.shutdown();
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for join operations: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static void testUpdatePerformance() {
  ExecutorService executor = Executors.newFixedThreadPool(10);
  long start = System.currentTimeMillis();
  for (int i = 0; i < 10; i++) {
    executor.submit(
        () -> {
          int iterationCount = 0;
```

```
        while (iterationCount < 5000) {
          Order order = getRandomOrder();
          order.setStatus("In work");
          orderService.updateOrder(order);
          ArrayList<OrderItem> orderItemsByOrderId =
              (ArrayList<OrderItem>) orderItemService.getOrderItemsByOrderId(order.getId());
          orderItemsByOrderId.forEach(
              orderItem -> {
                orderItem.setQuantity((int) (Math.random() * 9) + 1);
                orderItemService.updateOrderItem(orderItem);
              });
          iterationCount++;
        }
      });
  }
  executor.shutdown();
  try {
    executor.awaitTermination(5, TimeUnit.MINUTES);
    long time = System.currentTimeMillis() - start;
    System.out.println("Time for update: " + time + " ms");
  } catch (InterruptedException e) {
    throw new RuntimeException(e);
  }
}

private static Product createProduct(int i) {
  Product product = new Product();
  product.setName("Product " + i);
  product.setDescription("Product description " + i);
  product.setPrice(i);
  return product;
}

private static User createUser(int i) {
  User user = new User();
  user.setName("Name " + i);
  user.setLastName("LastName " + i);
  user.setAddress("Address " + i);
  user.setEmail("Email " + i);
  return user;
}

private static User getRandomUser() {
  User user = null;
  while (user == null) {
    long userId = (long) ((Math.random() * (userService.findMaxId() - 1)) + 1);
    user = userService.getUserById(userId);
  }
  return user;
}

private static Order getRandomOrder() {
  Order order = null;
  while (order == null) {
    long orderId = (long) ((Math.random() * (orderService.findMaxId() - 1)) + 1);
    order = orderService.getOrderById(orderId);
  }
  return order;
```

```
  }

  private static Product getRandomProduct() {
    Product product = null;
    while (product == null) {
      long productId = (long) ((Math.random() * (productService.findMaxId() - 1)) + 1);
      product = productService.getProductById(productId);
    }
    return product;
  }

  private static OrderItem getRandomOrderItem() {
    OrderItem orderItem = null;
    while (orderItem == null) {
      long orderItemId = (long) ((Math.random() * (orderItemService.findMaxId() - 1)) + 1);
      orderItem = orderItemService.getOrderItemById(orderItemId);
    }
    return orderItem;
  }

  private static Order generateTestOrder() {
    Order order = new Order();
    order.setUser(getRandomUser());
    order.setDate(LocalDate.now());
    order.setStatus("New");
    return order;
  }

  private static OrderItem generateTestOrderItem() {
    OrderItem orderItem = new OrderItem();
    orderItem.setOrder(getRandomOrder());
    orderItem.setProduct(getRandomProduct());
    orderItem.setQuantity((int) (Math.random() * 9) + 1);
    return orderItem;
  }
}
```

ДОДАТОК Б

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	Опис
Пояснювальні документи	
Кваліфікаційна робота Батальський.docx	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Кваліфікаційна робота Батальський.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF.
Програма	
Батальський.rar	Архів. Містить коди програми і скомпільовану програму.
Презентація	
Батальський.pptx	Презентація кваліфікаційної роботи.